AFIT/GCS/ENG/92D-22

X-AAARF
An X Window Based Version
of the
AFIT Algorithm Animation Research Facility

THESIS

Charles R. Wright, Jr
Captain, USAF

AFIT/GCS/ENG/92D-22

AFIT/GCS/ENG/92D-22

X-AAARF

An X Window Based Version

of the

AFIT Algorithm Animation Research Facility

THESIS

Presented to the Faculty of the School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Science

Charles R. Wright, Jr, B.S.E.E

Captain, USAF

December, 1992

Acknowledgements

This thesis would not have been possible without the support and understanding of my best friend, my wife Amanda. That is a debt I will happily pursue the rest of my life.

Of course, Dr. Lamont deserves "extra credit," for his tolerance and understanding, and for laughing at my jokes and agreeing with nearly everything I said – no one else does that.

Special thanks to my friends in the Software Engineering sequence for allowing me into their presence and onto their workstations, and especially for so graciously tolerating an "old" infidel. (Thank you, Mary Anne, for sharing my views on the importance of ice cream.)

Thanks to Lt Col Amburn for letting me "keep my fingers in" and for answering questions out of turn.

Thanks also to Paul Chase for his gallant efforts at keeping me honest with his unending stream of questions and sometimes ecletic observations on AAARF. No one can understate the obvious better than an Aussie.

I must also acknowledge the contributions made by my sister Lydia, for keeping the welcome mat out, and her husband Joe, who always kept a seat open in the canoe, regardless of the weather.

Finally, Watson deserves honorable mention because he was always ready to play, no matter what time of night I wandered in.

Charles R. Wright, Jr

Table of Contents

	Page
Acknowledgements	ii
Table of Contents	iii
List of Figures	vii
Abstract	viii
I. Introduction	1-1
1.1 Background	1-1
1.1.1 Algorithm Classes	1-2
1.1.2 Algorithm Classes Supported	1-3
1.1.3 Animation Support	1-3
1.1.4 The AAARF Design	1-3
1.1.5 AAARF History	1-4
1.2 Problem	1-4
1.2.1 Outdated GUI	1-4
1.2.2 Parallel Algorithm and Performance Animation	1-5
1.2.3 Parallel Architectures	1-5
1.3 Scope	1-6
1.4 Assumptions	1-6
1.5 Overview	1-7
1.6 Summary	1-7
II. Requirements Analysis	2-1
2.1 Introduction	2-1
2.2 AAARF Status	2-1

Page

2.3 Summary of Candidate Tasks 2-3

2.4 Summary of Current Knowledge 2-4

2.4.1 Animation Systems - A Sampling of Representative Systems 2-5

2.4.2 Parallel Performance Animation Systems and Techniques . . 2-7

2.4.3 Conclusion 2-14

2.4.4 XWindows GUI Systems 2-14

2.5 Requirements 2-15

2.5.1 GUI Requirements 2-15

2.6 Summary 2-16

III. AAARF Operational Maintenance 3-1

3.1 Introduction 3-1

3.2 Problems with the AAARF System 3-1

3.2.1 AAARF in General 3-1

3.2.2 Parallel Computer Performance Monitoring 3-3

3.3 Recommended Changes 3-8

3.3.1 AAARF in General 3-8

3.3.2 Parallel Computer Performance Monitoring 3-8

3.4 Implementation and Analysis of Results 3-12

3.4.1 AAARF in General 3-12

3.4.2 Parallel Computer Performance Monitoring 3-13

3.5 Summary 3-17

IV. X-AAARF - Design and Implementation 4-1

4.1 Introduction 4-1

4.1.1 A Prototype X-AAARF 4-1

4.1.2 SunView in an OpenWindows Environment 4-2

4.2 Selecting a Replacement Graphical User Interface (GUI) - XView . . 4-3

Page

4.2.1 Analysis of X Window Development Enviornments 4-3

4.2.2 Motivations for Choosing XView 4-8

4.2.3 A Closer Look at XView 4-9

4.3 GUI Replacement - Design, Implementation and Results 4-11

4.3.1 Replacement Strategy 4-12

4.3.2 Test Strategy 4-14

4.3.3 Design/Implementation Issues 4-14

4.3.4 The AAARF Main Process 4-18

4.3.5 The Common Library 4-19

4.3.6 The Array Sort Class 4-21

4.3.7 The PViews Library 4-24

4.3.8 General Results 4-28

4.4 Summary 4-29

V. Conclusions and Recommendations 5-1

5.1 Conclusions 5-1

5.2 Recommendations 5-3

5.2.1 AAARF Maintenance 5-3

5.2.2 AAARF Training 5-4

5.2.3 Individually Windowed Views 5-4

5.2.4 AAARF as a Classroom Tool - The Client Programmer Interface 5-5

5.2.5 A Formal Specification Language for Algorithm Animation . 5-6

5.2.6 AAARF Responsibility 5-9

5.2.7 The Future of AAARF 5-9

Appendix A. A BRIEF Discussion of X A-1

Page

Appendix B. A Simple Example of PRASE Instrumentation - The Ring Program . B-1

B.1 Introduction B-1

B.2 Overview B-1

B.3 Running the Animation B-2

B.4 Bailing Yourself Out B-7

B.5 Instrumenting a Simple Cube Program B-8

B.5.1 Instrumenting the *host* Program B-8

B.5.2 Instrumenting the *node* Program(s) B-9

B.5.3 Changes to the Makefile B-11

B.6 Source Listings B-12

Bibliography BIB-1

Vita VITA-1

List of Figures

Figure — Page

3.1. Network Connections for Automatic Mode. In manual mode, the *fork/exec* link between *server* and *algorithm* is not present. The algorithm must be started manually. 3-4

4.1. Programmer view of the complete X Window System [31:12]. 4-4

4.2. The AAARF modular design. Each outer box is a self contained system with it's own window based interface. Inner boxes are separate processes. 4-13

4.3. (a) Original structure of the Main window. Each window is a separate window. (b) Original structure of the Algorithm window. The view windows are contained within the Algorithm window. The remaining windows are separate windows. . . 4-15

4.4. New structure of the Algorithm window. 4-16

4.5. SunView Master Control Panel for the ArraySorts class. 4-22

4.6. XView Master Control Panel for the ArraySorts class. 4-23

4.7. Master Control Panel for the Parallel Performance Class. 4-25

4.8. Parallel View Options panel. 4-26

5.1. Proposed environment for the development of a formal specification language for algorithm animation. 5-7

Abstract

Algorithm animation is the process of graphically representing the state changes which occur during the execution of the control structure (algorithm) of computer programs. Rather than simply viewing program execution as changes in the contents of static data structures, algorithm animation presents program execution as a series of state transitions. This is done by associating specific decision points, and their resulting actions during execution, with corresponding changes in the graphical representation of the algorithm's data structure. In effect, algorithm animation attempts to show the *why* (algorithm execution) that is associated with the *what* and *how* (changes in the contents of the data structures).

The AFIT Algorithm Animation Research Facility was developed by the Air Force Institute of Technology (AFIT) as a teaching aid for data structures and algorithm design of sequential processes. However, AAARF's unique design makes it particularly suitable for the animation of algorithms running on remote systems. In particular, an extensive set of parallel performance animations has been developed for the Intel iPSC/2 Hypercube for parallel program analysis and performance optimization.

The AAARF system was originally developed using the Sun Microsystems' SunView^{TM} windowing system. Recent advances in Graphical User Interface (GUI) technology combined with Sun's adoption of the X Window System as their workstation window environment, has necessitated the replacement of AAARF's GUI with a modern, X-based user interface.

This report describes the GUI replacement process, starting with selecting a GUI toolkit, designing and implementing the new user interface, testing, and finally the results of implementing the new user interface. Also included is a discussion of several changes/enhancements to AAARF which were necessary before the GUI replacement process began.

X-AAARF

An X Window Based Version

of the

AFIT Algorithm Animation Research Facility

I. Introduction

Algorithm animation is the process of graphically representing the state changes which occur during the execution of the control structure (algorithm) of computer programs. Rather than simply viewing program execution as changes in the contents of static data structures, algorithm animation presents program execution as a series of state transitions or *Interesting Events (IE)* [4:13, 56-57]. This is done by associating specific decision points, and their resulting actions during execution, with corresponding changes in the graphical representation of the algorithm's data structure. In effect, algorithm animation attempts to show the *why* (algorithm execution) that is associated with the *what* and *how* (changes in the contents of the data structures).

Algorithm animation is useful in the development of new software, as well as the effective use of existing software. It is also an effective aid in teaching algorithm design and understanding the behavior of existing algorithms. Recently, algorithm animation techniques have been adapted for use with parallel computers [29] as an aid in understanding the complex inter-relationships that exist between parallel computer architectures, and the decomposition and partitioning of algorithms for execution on these machines.

1.1 Background

The AFIT Algorithm Animation Facility (AAARF) [4, 29] is a general purpose visualization tool for the animation of algorithms. The applications for AAARF are education, and analysis and

debugging. To support these applications AAARF provides for two classes of users: *end-users*, and *client-programmers* [4:1][2:6].

In the educational mode, end-users interact with animations developed by client-programmers for the purpose of studying and analyzing the execution of a particular algorithm or class of algorithms. In the client-programmer mode, the users are, typically, the client-programmers themselves. Client programmers use AAARF for two purposes: the development of new classes for end-users, and for debugging and analysis. AAARF provides two user interfaces: a window based, mouse driven interface for end-users, and a function and library based software interface for client-programmers. AAARF is written in C [12] and uses the XView (X Window-System-based Visual/Integrated Environment for Workstations) [10] user-interface toolkit from Sun^R Microsystems, running under OpenWindows^{TM} [27]. AAARF currently runs on Sun SPARCstation2 workstations. AAARF supports the animation of processes running on both serial and parallel architectures.

1.1.1 Algorithm Classes AAARF partitions algorithm space into classes, with class membership being defined as a function of the transformation process an algorithm performs while operating on an input set and producing an output set. (This classification scheme is defined formally [4:11-14].) This classification scheme can be applied broadly, as in the case of integer array sort algorithms, or more narrowly, as a measure of the effectiveness and efficiency of different implementations of a particular algorithm. In the broad sense, algorithms from the same class are defined to operate on the same input, and produce identical outputs, albeit by different means. In the narrow case, the same algorithm operates on an input, producing two kinds of output which are of interest: that which is specific to the algorithm (and its associated "broad" class) and implementation specific information (usually architecture related), from which measures of efficiency and effectiveness can be made. In the narrow case, interest is in the architecture specific information because it allows the comparison of various algorithm implementations. This is important in the case of parallel computer architectures, since an algorithm may be implemented in a vari-

ety of ways, with the preferred implementation being determined only after exhasutive empirical testing. Clearly, this classification scheme is not exclusive, since comparisons can be made across implementations of algorithms from within the same class.

1.1.2 Algorithm Classes Supported AAARF currently supports three serial algorithm classes: 1) integer array sorts (with nine sorting algorithms), 2) tree traversal, and 3) dynamic tree searching, with an implementation of the traveling salesman problem as an example algorithm. For parallel architectures, AAARF provides a set of 11 animations for performance monitoring of the Intel iPSC/2.

1.1.3 Animation Support AAARF provides a broad spectrum of support for algorithm animation. As stated above, the AAARF end user interface is window based and mouse driven. These kinds of interfaces are generic1y referred to as Graphical User Interfaces, or GUIs. The AAARF GUI provides for the display of multiple views of algorithms as well as the simultaneous display of multiple algorithms. The ability to save and restore the current configuration is also provided, along with an animation record capability for playback at a later time. A full range of algorithm and configuration control facilities are also provided, such as variable speed control, breakpoint selectors, view configurations, etc [6]. The client-programmer interface is a rigid, parameterized software interface. It consists of a set of predefined functions which the programmer "fleshes out" along with a collection of AAARF library routines [5]. Client programmers do not add graphics to their computer programs. Instead, they design a set of algorithm (or class) specific graphics routines which are then added to AAARF's display facilities. These routines are called in response to a timer which controls the animation by periodically querying the algorithm for ... next *IE*.

1.1.4 The AAARF Design The AAARF design is that of multiple, cooperating processes [Ous] which communicate via Unix sockets [21]. The advantage of this configuration is that it is not necessary for the display process to be on the same machine as the algorithm being animated.

This is exactly the technique used to animate algorithms running on parallel architectures. The disadvantage, of course, is that it is necessarily more complicated, and fragile.

1.1.5 AAARF History AAARF was designed and implemented as an animation system for serial processes by Fife [4] in 1988-1989. It was extended by Williams [29] in 1989-1991 to include animations for parallel performance analysis and parallel algorithms on the Intel iPSC/2 Hypercube. M.D. Lack [13] further extended AAARF during 1990 - 1991 by adding an expert system advisor and additional iPSC/2 animations. AAARF contains between 15,000 and 20,000 lines of source code. The entire system, compiled and uncompressed, occupies 25 megabytes of disk space.

1.2 Problem

The orignal intent for AAARF was to provide a platform for algorithm research. This has never really transpired, for several reasons. First and foremost was a lack of available workstations. This problem has recently been addressed with the purchase of some forty Sun Sparcstation2s for use by the general AFIT engineering student body. Since these workstations run Openwindows, new students are indoctrinated into the Openwindows environment, and are therefore unfamiliar with Sunview.

The original AAARF design was done using modern object-oriented techniques [4:36-46]. This design is still valid: indeed, as the capabilities of modern workstations have expanded, it has actually become more relevant. (This is a rare phenomenon in the world of computers and computer software!) Certain facets of the design's implementation, however, have become outdated. These are outlined below.

1.2.1 Outdated GUI The original AAARF system's window and mouse driven interface was written in C and used the SunView [25] windowing environment. Sun has replaced SunView

with OpenWindows (an X Window System based GUI). OpenWindows follows the OpenLook^{TM} protocol, which supports the messy desk environment. This environment is much less rigid than the SunView environment. However, the overwhelming consideration is that the SunView and OpenWindows environments operate independently of one another. This results in two separate, incompatible window managers trying to manage a single screen. For example, SunView windows cannot be moved forward or backward of OpenWindows windows, and vice-versa, because the two window managers are not aware of each other, much less each others windows. Furthermore, Sun has indicated that support for SunView will soon be phased out, further emphasizing the need to update AAARF. All of this adds up to a system that has an unfamiliar interface, and is difficult to use, especially for new users who have no previous experinence with SunView.

1.2.2 Parallel Algorithm and Performance Animation Using AAARF in the parallel performance mode is best described as difficult. This stems partly from the outdated Sunview GUI system discussed above, and partly from the inherent fragility of a system composed of multiple processes communicating over a network. There is also a definite lack of quality documentation for the instrumentation process. AAARF needs to be made easier to use in this mode to make it more useful as a parallel algorithm analysis and debugging tool. It is easy for experienced programmers to lose sight of fact that many users are naive and do not experiment - they learn how to do something one way and never consider or explore alternatives. With AAARF's current configuration, such an approach is simply not possible. Users must be creative and determined or they will have a difficult time with AAARF when animating parallel algorithms.

1.2.3 Parallel Architectures AAARF currently supports the Intel iPSC/2 Hypercube for parallel algorithm animation. This architecture is being phased out and support will be discontinued at the end of calander year 1992. AFIT has access to other parallel architectures, most notably the iPSC/860. It will be necessary to add the iPSC/860 to the list of parallel machines supported by AAARF.

1.3 Scope

For AAARF to truly become what its designers had envisioned, it will be necessary to make AAARF easier to use and more reliable, particularly in the case of parallel algorithm animation. The intent is to evaluate AAARF's current communications protocols, with the goal of removing any reliance AAARF places on the user for providing run time configuration information. In part to satisfy the above requirements, and to make AAARF more readily available, it will be necessary to replace the current SunView based GUI with a new, X based GUI. The goal is a new X based AAARF which is as reliable as the current configuration, easier for the end-user to use, and with a simpler client programmer interface.

The problems outlined previously in Section 1.2.1 regarding AAARF's GUI are not yet critical, which is precisely why now is an appropriate time to address them. As stated earlier, AAARF is quite large, and growing larger with each thesis cycle.. The task of replacing the GUI is within the scope of one thesis cycle, providing certain restrictions are given due consideration. These are discussed in detail later in the report, but center around how much of the core event-handling structure must be modified, and or replaced, to accommodate the new GUI. If the core event handling structure cannot be maintained, it may then be necessary to completely rewrite AAARF - which is clearly beyond the scope of one thesis cycle. (A prototype AAARF with an X based GUI was done by Williams in Jan-Feb 1991 and extended by Lack [13] in late 1991. Results are discussed in Chapter IV).

1.4 Assumptions

Currently, AAARF is primarily used as an analysis and debugging tool for programs written for the iPSC/2 Hypercube. Since the problem of workstation availability has eased, it is expected that the use of AAARF as an instructional aid will increase significantly during the next year. Furthermore, AFIT appears to have standardized on the Sun Sparcstation platform as the engineering

workstation of choice. It is expected that the user community will remain AFIT for the foreseeable future, which means that portability is currently not a major issue.

1.5 Overview

This investigation is divided into three parts:

1. Part 1 is a requirements analysis, including a review of current literature, with emphasis on parallel algorithm animation and parallel program performance analysis. This is presented in Chapter II.
2. Part 2 is an analysis of AAARF to determine what steps can be taken to simplify the use of AAARF before the GUI replacement phase begins. This is necessary for two reasons: 1) to provide a simpler to use platform for students using AAARF in support of research, and 2) to provide a stable platform, not subject to constant change, while phase 3 is underway. This is Chapter III.
3. Part 3 is a rewrite of the AAARF GUI. Phase 3 starts with an analysis of currently available GUI systems and their attractiveness to AAARF. This is followed by a detailed discussion outlining the motivations for choosing XView. Part 3 is presented in Chapter IV.

1.6 Summary

The goal of this thesis effort is to make AAARF a more stable and easier to use platform for algorithm animation. The following chapters chronicle the changes made to AAARF to enhance its usefulness as an educational, and analysis and debugging tool.

II. Requirements Analysis

2.1 Introduction

The goal in requirements analysis is to decide what is to be done. As is usually the case in an academic environment, there is more to do than the allotted time allows. With this in mind it is necessary to prioritize the list of potential tasks and choose those which satisfy the immediate needs of AAARF within the time allotted. Chapter II starts with an analysis of the status of AAARF prior to the start of this thesis cycle. Next is a summary of candidate tasks, presenting and justifying what was chosen, followed by a current literature review. Chapter II finishes with a "formal" requirements statement.

2.2 AAARF Status

As stated in Section 1.2, AAARF was designed using modern object-oriented techniques. Both the design and the implementation remain valid. The implementation, while not object-oriented, is modular and maintainable, and has robust, production code quality, error handling facilities. There is good overview documentation, but there is a corresponding lack of detailed implementation information. The source code contains marginal comments of the type typically found in computer programs: they describe what a module does in general terms, but lack specific information regarding how or why.

Because of its reliance on SunView, AAARF is not portable across architectures (recall from Section 1.1.4 that AAARF can animate processes running on other architectures). AAARF is a multi-process system. This complicates serious error detection and recovery, and it also complicates analysis and debugging. The network and inter-process connection scheme used for animating parallel (iPSC/2) programs is very fragile (this is covered in detail in Chapter III, see Figure 3.1, page 3-4). The process for running animated parallel programs is strictly scripted and cannot be deviated from. Running AAARF successfully requires an above-average-knowledge of Unix,

especially if errors occur. For the client-programmer, extensive knowledge of C, file descriptors, Unix Sockets and, especially, the Unix make facility are necessary. Obviously, there is a huge learning curve. (In all fairness, AAARF is a very well written system. AAARF was written by two programmers who have well above average programming capabilities. As a result, AAARF contains coding techniques, and makes use of Unix facilities, which would not normally be found in graduate thesis code.) The learning curve, combined with a lack of quality documentation, forces potential users to rely heavily on the local AAARF expert. As expected, this can put a serious strain on that resource. Perhaps the single largest problem is that of adding a new class to AAARF. The prospective client programmer must know C, the Unix make facility, and he/she must understand enough about AAARF to know where to look for direction. Unfortunately, this task is beyond what can normally be expected during a quarter. This single factor, above all others, has limited the spread of AAARF.

At the beginning of this investigation, AAARF was available on six workstations, two Sun3s and four Sun4s. There was, however, no fully functional, running version of AAARF. There are two reasons why this happened: the first has to do with the manner in which AAARF is maintained; the second has to do with the instability of the workstation environment at AFIT. In the first case, the only copy of AAARF available is maintained by the local AAARF expert (the individual chosen to have AAARF as his thesis topic) somewhere off of his login directory. These individuals typically have multiple copies (with no supporting documentation as to their respective states), some, none, or all of which could be usable at any given point in time. The second reason revolves around the constant changes in the workstation environment. These changes are primarily due to a shortage of mass storage space. Users are moved about from one file server partition to another by the system administrators, as dictated by space requirements. Since AAARF is run-time dependent upon certain path information contained in files, such a move can render AAARF unusable until this information is updated. For the experienced AAARF expert this is only an annoyance. For the naive AAARF user, this is just enough of a problem to cause them to stop using AAARF.

Operating system upgrades, window system upgrades, networking changes, and the like also pose occasional problems as well. For a combination of the reasons stated above, there were actually three copies of AAARF available at the beginning of this investigation, none of which were fully functional.

2.3 Summary of Candidate Tasks

A number of opportunities exist for enhancing and extending the capabilities of the AAARF:

- Make AAARF a single-copy multi-user platform.
- Improve the client-programmer interface by simplifying it and providing detailed documentation with appropriate examples.
- Perform a capability analysis of existing animation packages for parallel architectures. The purpose here is to provide direction, prevent duplication, and uncover potential dead ends.
- Extend the capabilities of the current expert advisor.
- Provide a GUI interface to the expert advisor.
- Replace the SunView GUI.
- Port the parallel instrumentation facilities to the iPSC/860
- Add sound generation to the animation facility. This will require an analysis of the current state of the art and its potential application to AAARF.

Before work on the chosen tasks can begin, it is necessary to make the current AAARF system functional. Because item 2 above is considered a relatively straightforward task, it is done in conjunction with this step. Selection of the remaining tasks will be done in parallel with the above task. The selection criteria are listed in order of importance below:

- Maintaining AAARF in a stable condition,
- Enhancing the attractiveness and ease of using AAARF,
- Improving the functionality of AAARF,
- Extending the capabilities of AAARF.

The single overriding factor in choosing which tasks to pursue is the need to keep AAARF running and available for future students and AAARF researchers. Obviously, consideration must be given to those tasks which, if ignored or postponed, pose the most immediate threat to the stability of AAARF. Once the stability of AAARF has been ensured, attention can be focused on improving the client-programmer interface. Two tasks meet the stability test: replacing the SunView GUI, and porting the parallel instrumentation facilities to the iPSC/860. These two are considered the most important and are the subject of this investigation. Since a current literature review is part of any thesis effort, the capability analysis is included by default. A review of currently available GUI systems applicable to AAARF is included as part of the GUI replacement task.

2.4 Summary of Current Knowledge

Algorithm animation and program visualization are techniques of graphically representing the execution of programs. Algorithm animation is a relatively new field in computer science. The term program visualization is applied to the graphical representation of data generated by programs. It aids users in understanding the data, especially data with complex structures and inter-relationships. Program visualization techniques have been used effectively for many years and are an integral part of many successful software packages. Recent emphasis is in providing users (programmers) with a set of standard interface routines for graphically representing program execution results. Both algorithm animation and program visualization are vital techniques because humans rely heavily on mental images for problem solving. While this review concentrates primarily on algorithm animation, it must be noted that the animation of algorithms relies heavily on program visualization techniques, making it difficult to talk about one without considering the other.

The definition of algorithm animation is abstract enough to allow the inclusion of systems which, at first glance, might not appear to be algorithm animation systems. Because this is best

illustrated by example, two systems, the Visual Programmers Workbench and the Visualization and Interactive Programming Support system, are included in the review.

The review that follows first explores several representative algorithm animation systems to give the reader insight into the focus and direction of current research and how such systems are employed. The next section explores animation systems and/or techniques specific to parallel computer architectures. Finally, the last section discusses salient issues common to all parallel algorithm animation systems, including the addition of aural cues and instrumentation issues.

2.4.1 Animation Systems - A Sampling of Representative Systems Recent interest has centered around simplifying the use of algorithm animation systems to make them more generally accessible to non-expert or casual users. This stimulates the use of these systems and promotes the application of these tools to algorithm design and analysis. Several recently described systems are discussed with the intent of giving the reader a general feel for the state of the art, as well as some indication of the direction that future algorithm animation research might take.

2.4.1.1 TANGO - Transition-based ANimation GeneratiOn Most algorithm animation facilities require users to design their own animations using the graphics libraries on the host system. This can be (and usually is) a daunting task. Perhaps the most important aspect of encouraging more general use of algorithm animation systems is to free users from the burden of developing their own animations [20]. TANGO is an animation facility that attempts to relieve programmers of this burden. TANGO is built on a framework which allows users to develop sophisticated, real time animations without low-level graphics coding [20:1]. This is accomplished by abstracting the animation portion from the program being animated. A simple but powerful set of data structures and operations allows the programmer to interface with the animation package and to associate events or actions in the program with the location and movement of graphic entities on the screen. The path-transition paradigm [20:2] provides a mechanism for displaying the fluid movement of graphic entities during state changes. TANGO can also be used to animate paral-

lel algorithms since it provides the capability to drive a single animation from multiple executing processes. TANGO is an X11 based application and runs on Sun and DEC workstations.

2.4.1.2 VIPS - Visualization and Interactive Programming Support VIPS is a graphical extension to UNIX's symbolic debugger, DBX. It is a linked list visualization tool for debugging. It can dynamically display linked list structures, portions of linked lists, and the fundamental (underlying) structure of large, complex linked lists [19:4-5]. It also has the ability to identify which nodes are changing location or contents during execution. While VIPS is not intended to be an algorithm animation facility, it is a good example of how program visualization techniques can provide valuable insight into the operations of computer programs. The graphics facilities and debugging capabilities of VIPS would be a valuable addition to any algorithm animation package.

2.4.1.3 GAIGS - Generalized Algorithm Illustration through Graphical Software GAIGS is an instructional system used for classroom support. The motivation behind GAIGS is to relieve students of the burden of the graphics programming so that they can concentrate on drawing conceptual conclusions from the results of their algorithm programming efforts [15:105]. GAIGS distinguishes between an algorithm's implementation, which is any program that results in the execution of the algorithm, and its visualization which is a sequence of graphic snapshots that represent the algorithm's operation upon data structures [15:106]. GAIGS is a library of animations for specific algorithms which are frequently taught in computer science courses. GAIGS reads a text file of animation commands generated by the users program and translates them into the appropriate graphics for animation. No knowledge of graphics programming is required on the part of the user. GAIGS is a static animator in that it can only show the effect of the algorithm's execution upon data structures; it cannot show state transitions or events. Any language capable of generating text files which follow the GAIGS protocol can make use of GAIGS. GAIGS is not a real time animation system since the algorithms must first be run to generate the input file for GAIGS.

2.4.1.4 VPW - The Visual Programmers Workbench Visual programming languages use graphical means for representing program objects and allow these objects to be arranged on the screen in a two dimensional way [28]. The next logical step is an environment for the construction of visual programming languages. VPW is such an environment [18]. The VPW environment consists of the following types of specifications: the syntactic structure, the abstract structure, the static semantics and the dynamic semantics [18:553]. The syntactic structure specifies the visual appearance and structure of the language. The abstract structure defines a model of the language's structure. The static semantics specify the static properties of the language (for example, type checking) while the dynamic semantics describe the execution properties. Using these four specifications it is possible to completely describe and implement a visual programming language. VPW is not an algorithm animation facility; VPW uses graphical objects to construct algorithms rather than graphically depicting the execution of algorithms. However, since VPW provides a "built in" mapping of graphical objects to executable code, it can function as an algorithm animator. In this regard, VPW (and other visual programming languages in general) is a very powerful tool because it is very easy to make program changes and observe the consequences.

2.4.2 Parallel Performance Animation Systems and Techniques The performance of parallel algorithms is heavily dependent upon both the method used to partition the algorithm for execution, and the target machine architecture. For these reasons, the primary use of animation in parallel computers is performance animation rather than actual algorithm animation. The technique of animating algorithms, that of instrumenting the source code and animating events, is readily adapted to parallel performance monitoring. Examples of typical events are the sending and receiving of messages, locking and unlocking of shared resources, etc. The fact that the same techniques used to monitor and measure the efficiency and effectiveness of algorithms executing on sequential processors can be applied to the performance of algorithms executing on parallel processors is extremely important: one animation system suffices. (Remember that on parallel

architectures the interest is usually on *how* an algorithm is partitioned for execution and not *which* algorithm is being used.)

A detailed state-of-the-art analysis of performance and animation systems was done by Lack [13] in '91. Many of the systems reported on by Lack were architecture and/or application specific. Recent emphasis focuses on portability for the animation systems and standards for the instrumentation trace data. For this reason, only those systems which have demonstrated portability and/or have been adopted as a quasi standard by the research community are reported on here. The following sections give an overview of these systems in moderate detail including several new entries in the field. Also included are several research efforts which are currently using these systems. This section concludes with a discussion of general trends in parallel performance and algorithm animation.

2.4.2.1 PICL - Portable Instrumented Communication Library PICL is a portable instrumented communication library designed to provide portability, ease of programming, and execution tracing in parallel programs [8]. Obviously, PICL is not a animation system. Since all animation systems require some form of trace data, it is included here as a representative of the class of programs which collect data for performance analysis purposes. PICL was initially developed as a portable communications library for distributed memory parallel multi-processors. The library consists of 12 low-level communications and system interface routines, and 14 high-level routines which implement commonly used parallel architecture functions (global broadcast, barrier synchronization, etc.). Eventually, execution tracing facilities were added. It is the execution tracing facilities which are of interest. There are 9 execution tracing routines. An interesting aspect of the PICL trace facilities is that users can specify the level of observation to use in monitoring execution, and they can change the level during execution. The obvious advantage is that monitoring can be tailored to focus on specific areas of interest. PICL's monitoring facilities generate two types of trace data: an augmented format to enhance human readibility, and a compact

format intended for use with the ParaGraph [9] algorithm animation system. In addition, PICL allows users to define "task" specific trace records which can be used to logically mark where in the program particular behavior(s) occur.

2.4.2.2 ParaGraph With 25 predefined performance animations, ParaGraph is easily the most "comprehensive" parallel performance animation system available. ParaGraph is a portable, post processing system. Paragraph gets it's portability from PICL, the source for it's trace data, and X Windows, it's GUI system. ParaGraph runs on both color and monochrome displays, but the animations are most informative when viewed in color. ParaGraph uses many of the same display formats as AAARF (such as Gantt, Animation, Kiviat, time-space or Feynmann, message and communications load, etc.). ParaGraph also provides several displays which are currently not implemented in AAARF, one is a variation on the space-time plot which shows the longest serial thread running through the execution and the other two are phase plots which show the relationship over time between communications and processor use [9:37].

A very interesting aspect of ParaGraph are the "task" displays. While the standard displays are informative and useful, they contain no information about where in the program the events are occurring. This is partially solved by allowing users to logically define "tasks" by bracketing specific sections of code with the PICL task begin and end records and assigning it a task number. Numbers need not be unique to a processor, thus tasks can be spread over multiple processors (which is the essence of parallel processing anyway). There are several task displays. In these displays, each task is assigned a different color, thus allowing users to see where in the run specific sections or lines of code are being executed. These displays, when combined with the standard displays, help provide a more complete and accurate picture of program execution.

ParaGraph is extensible, allowing users to develop their own performance displays; at appropriate points calls are made to user supplied routines for the initialization, data input, event handling, and drawing of application specific animations [9:38].

Paragraph is not an algorithm animation system, although there is nothing in the design of ParaGraph to preclude this. The problem is actually with PICL: there is no formal mechanism to allow users to define their own trace record formats, which would be necessary for algorithm animation. Both systems are distributed with source code, so it seems likely that ParaGraph combined with PICL and the necessary modifications to both could produce an algorithm animation system.

2.4.2.3 VISTA - Visualization and Instrumentation of Scalable multiComputer Applications VISTA is not an animation system, rather it is an instrumentation and visualization paradigm [3:1] intended to solve the scalability problem inherent in most performance animation systems. The VISTA paradigm treats performance data essentially the same as distributed data in the context of the programming models used for parallel programming. This amounts to data-parallel mapping of program onto machine and allows the performance to be viewed as it relates to each processor, processor cluster, or the processor ensemble and as it relates to the data structures of the program [3:1].

The VISTA paradigm is composed of three components, Visualization, Data Parallel Representation, and Performance Measurement. Of particular interest is the Visualization component. The basis for the Visualization component is the state of a Processing Element (PE), which translates to one or more quantitative metrics K (a scaler, either measured or calculated, this is formalized by the Performance Measurement component). The visualization Component is divided into four levels:

- **microscopic snapshot** A performance parameter K at some specific time on a particular processor.
- **microscopic profile** A microscopic snapshot which allows K to vary over time. An AAARF or ParaGraph style single processor display.
- **macroscopic snapshot** A microscopic snapshot of all PEs at a specific time for a particular K. This forms a two dimensional mapping and is essentially the same idea used by AAARF and ParaGraph for multiprocessor displays.
- **macroscopic profile** A macroscopic snapshot allowing K to vary over time.

The four levels are referred to as Machine Views, and are instances of a general class of multivariate data plots tailored to display performance measurement data [3:5] . Macroscopic views correspond to images, thus image analysis and multivariate statistical analysis techniques are used for interpreting the data [3:6]. This sounds rather sophisticated, but in reality, many of the performance views provided by systems like AAARF and ParaGraph satisfy the "image" label.

The VISTA paradigm has been implemented on an nCUBE2 using the PICL-ParaGraph [8, 9] system and on a MasPar MP-1 using the MasPar Programming Environment's Machine Visualizer Window [14]. Both animation systems have been extended to satisfy the macroscopic profile level of the VISTA visualization component. Several other visualization systems are mentioned, but no indication is given at to whether VISTA has been implemented using these systems.

VISTA exceeds the capabilities of current animation systems because the hierarchical levels of the Visualization Component scale up well beyond the 128-256 processor upper limit typically found in parallel performance visualization and animation systems. By looking at clusters and ensembles of processors as individual units, while still maintaining the ability to "look closely" with microscopic views at individual processors, the VISTA paradigm is the first performance animation system that is truly scalable.

2.4.2.4 Seeplex Seeplex is a real time parallel computer performance monitoring system to help programmers load balance an algorithm. Load balancing, a technique of distributing an algorithm's components in a parallel computer for maximum efficiency, is a common problem in developing parallel applications, and can often only be done empirically [9]. Seeplex, which runs only on the NCUBE parallel computer, gets it's data from the Simplex operating system, which has extensive instrumentation for performance monitoring. Seeplex provides an extensive and very flexible set of icon based tools which allows users to easily construct various views of system performance. Seeplex can be configured to monitor typical parallel computer performance

criteria (message traffic and message queues, node state information, etc.). The information displays generated by Seeplex are very similar to those produced by AAARF.

2.4.2.5 Los Alamos National Laboratory Hotchkiss and Wampler [11] have developed an algorithm auralization system. The premise for their work is that within as little as five years animation systems will not be capable of conveying the amount of information generated by massively parallel computers.

To date the authors have concentrated on auralizing mathematical functions, such as $y = xn$ and $f(t) = t + sin(wt)$. The authors use three basic parameters, frequency, amplitude, and time, to represent various situations which arise in evaluating mathematical functions. An interesting aspect of the authors work is that they make use of a variety musical instruments via a Yamaha synthesizer.

They have produced some very intriguing results:

> Chaotic functions such as the bifurcating function $x(i+1) * x(i)r$, when reiterated to convergence, clearly conveys the converged functional behavior as well as the chaos where convergence does not occur ... we have discovered that audibilized mathematical functions can create sounds that no man has ever heard before and truly excite even the non-musical mind.

Clearly, the use of sound to represent execution events or trends presents some interesting possibilities, especially when combined with animations. The authors freely admit, however, that defining what constitutes "good" performance or "accurate" results in terms of sounds is extremely difficult.

2.4.2.6 Trends in Parallel Performance Animation Current research in parallel performance animation is focused in two critical areas: visualization/animation techniques for massively parallel machines, and development of an execution trace standard or standards. Also, as the number of processing elements continues to grow, the amount of trace data generated grows

accordingly. Nearly all animation systems operate offline, so mass storage space while the trace data is being generated is also a problem. This, too, is related to trace standards, since any standard must compact trace data as much as possible. There appears to be little or no attention given to real time animation. The general consensus seems to be that communications and graphics processing bandwiths will never be wide enough to allow for real time animation of massivel, parallel computers. Obviously, humans are incapable of processing information quickly enough to make real time *performance* animation practical. However, real time *algorithm* animation is feasible and is something that has escaped scrutiny by the research community. For very large computer problems, such as the Grand Challenge problems, interactive real time control of the search process could be very useful in bounding the search space and significantly speeding up the search as well as potentially producing better solutions. Of course, until the communications and graphics bandwith problems are solved, this is just fringe thinking.

Francioni and Rover in [7] discuss the use of sound to relieve the problem of high density graphics when performance animating programs for massively parallel machines. The technique employed involves mapping one or more of the visual performance parameters to an aural based representation. The conclusion is that aural cues can enhance the speed of recognition and distinction of whole and partial programs [7:434]. It is not clear, however, that aural cues scale, or scale as well as visual cues since aural processing in humans is not as effiecient as visual processing. Nevertheless, it seems safe to conclude that a combination of visual and aural cues is more effective than either alone.

There are, of course, other issues. The effect of instrumentation on timing is always a concern. To date, the method of instrumentation varies widely from hardware support, to OS support, to intercepting function calls. All have advantages and disadvantages: trace files generated at the hardware or OS level are less intrusive but lack flexibility, while intercepting function calls usually requires modification of the source code, but is very flexible. Most massively parallel machine

vendors recognize the need for execution tracing and are built in support at one or more levels. For example, Intel's new Paragon machine covers both ends of the spectrum: they are providing hardware support if the form of a dedicated on board instrumentation processor running in parallel and software support by including (unsupported) the PICL-ParaGraph system of performance monitoring.

2.4.3 Conclusion The above reviews give a fair picture of the state of the art of algorithm animation and its applications. All of the systems are primarily research platforms. To my knowledge, no commercial system is available, although all of the systems reviewed are available to interested researchers. Several of the reviewed systems, such as Paragraph, and PICL are being incorporated into the operating systems of some of the new massively parallel machines. For example, Intel is including both Paragraph and PICL in its OSF operating system for the new Paragon massively parallel machine.

Much research still needs to be done. For example, there has been very little work done in studying how best to represent and display conceptualizations and other non-graphical information on computer screens. This is because the emphasis has been in building prototypes to demonstrate feasibility. The translation of algorithms into graphics, and its compliment, are not independent of the semantics of the problem being solved. This can make automating the process very difficult. If the development of algorithm animation systems continues to follow the general trend in computer software development, many point solutions will be demonstrated before a more general approach is adopted. Obviously, a truly useful programming environment would contain aspects of all of the reviewed systems.

2.4.4 XWindows GUI Systems This section is a brief description of each of the GUI systems available for the Sun Sparcstations. It is not a complete list and does not include, most

notably, user interface design tools.[1] It is assumed that the reader is familiar with the X Window System. If this is not the case, Appendix A contains a brief description of X with references. Detailed analysis of each GUI system can be found in Chapter 3. *Widget* is user interface parlance for object.

- *OLIT* is the OPEN LOOK^R Intrinsics Toolkit. It is Sun's and AT&T's implementation of the OpenLook GUI Standard [27]. It provides a robust, extensible set of predefined widgets. It is built on top of the Xt Intrinsics toolkit and Xlib.
- *OSF/Motif*^{TM} is the Open Software Foundation^{TM} (OSF) graphical user interface design toolkit [16]. Like OLIT, Motif provides a large set of commonly used, predefined, and extensible widgets. It is also built on top of the Xt Intrinsics toolkit and Xlib.
- *Athena* is the X Consortium's widget set. It follows no particular GUI standard. The widget set is considered weak in comparison to OpenWindows and Motif. Athena was originally developed in response to user complaints that a widget set was not included as part of the X distribution package.
- *XView* is Sun's attempt to provide backward compatibility to the large number of SunView applications that are still around (such as AAARF) [10:*xxiii*]. The object set is not as robust as either Motif or OLIT. XView's main attraction is that it transparently supports (most) SunView calls. XView is, with a few minor exceptions, OPEN LOOK compliant [10:669-673].
- *Xt Intrinsics* provides the basic user interface components from which most other user interface objects (widgets) are constructed [31:14]. It is generally considered too low level for user interface design. Xt is not evaluated in Chapter IV as a candidate GUI.
- *Xlib* is not really a GUI at all. Rather, it is the C language interface to the XWindows protocol (there is only one). It is too complex and too low level for serious consideration as a GUI language. All of the above systems, at one level or another, are implemented with Xlib. Xlib is not evaluated in Chapter IV as a candidate GUI.

2.5 Requirements

2.5.1 GUI Requirements This section presents a formal statement of the requirements which were used to make the final selection of the GUI system to be used for the SunView GUI replacement task. These are only the requirements. A detailed discussion of the pros and cons of each GUI as they relate to the requirements is presented in Chapter IV. Each of the requirements categories is listed with its corresponding rating, followed by the criteria used to derive the requirement(s).

[1] User interface design tools allow for the automatic construction of user interfaces using, typically, one of the GUI languages described above. They are, in effect, a graphical user interface for the construction of GUIs. These are useful in constructing user interfaces for new systems, but are not particularly useful in the case of AAARF.

- **Target User Group** – AFIT student body
  The target user group is divided into two categories: AFIT students using AAARF as an educational tool in support of classroom assignments, and AFIT students using AAARF as an analysis and debugging tool in support of research. Of the two, the later is more common. For the foreseeable future, the primary users are likely to be AFIT students.
- **User Friendliness** – End User: High, Client-Programmer: High
  From and end-user perspective, how intuitive are the interface objects, are their functions obvious, or is extensive training required? For the client-programmer, how good is the software interface, are data easy to get into and out of the objects, and is the object structure clearly defined? Is the needed functionality available as an object, or must it be built from lower level constructs?
- **Compatibility with the "Normal" Workstation Environment** – Desirable
  Will the GUI run under the OpenWindows environment, or will it be necessary to switch from OpenWindows to some other window manager? If it is necessary to switch from OpenWindows, how foreign will it appear to students familiar with OpenWindows and will the necessity to change limit its use?
- **Availability** – Necessary
  Is the GUI under consideration available for the Sparcstation2 and is there a cost and/or licensing fee associated with it?
- **Reliability** – Necessary
  What is this GUI's reputation for reliability, how often is it revised or updated, who is the vendor and what is their reputation? What is their relationship to Sun?
- **Portability** – Desirable
  Is this system portable to other architecture's? If so, what are the cost and licensing issues, and will it require changes to the basic AAARF design?
- **Enhanced Capabilities** – Desirable
  Will this GUI provide capabilities not available with SunView?
- **Development Effort/Time** – Low
  What is the effort involved in applying it to AAARF? How long will it take to replace the SunView GUI using this software? Can it be predicted accurately?
- **Procurement Effort** – Low
  What is the effort involved in procuring the software?
- **Procurement Time** – Short (already available preferred)
  How long will it take to get the software, and are there any special permissions or waivers required?

2.6 Summary

First and foremost, the current SunView version must be made runable in the current operating system and windowing environment. This is necessary for two reasons: first, to give students who wish to use AAARF during the current thesis cycle a stable platform; second, to provide a benchmark against which the new AAARF can be measured during development.

III. AAARF Operational Maintenance

3.1 Introduction

The SunView version of AAARF requires maintenance (Section 2.3) before work can begin on replacing the GUI. This is necessary for two reasons: to provide a stable platform for current students wishing to use AAARF in support of their research, and to provide a functioning benchmark for comparison purposes during the GUI replacement phase. In support of these requirements, Section 3.2 is a detailed investigation of known AAARF problems and weaknesses. The scope of this investigation exceeds what is required to satisfy the stable platform and benchmark requirements. The motivation for the expanded scope is to provide a broad discussion which can serve as both documentation and direction for future AAARF researchers. Recommended changes to AAARF which satisfy the maintenance requirements specified above are presented in Section 3.3. Section 3.4 is an analysis of the results of implementing those changes.

3.2 Problems with the AAARF System

AAARF's problems can be roughly divided into three categories:

1. problems that are generic or general in nature, e.g. not related to any particular feature or facet of AAARF;
2. problems specific to the SunView windowing system;
3. problems with remote animation, and in particular, the iPSC/2 Hypercube.

Only the first and last of these are included here, problems with the SunView windowing system are deferred to Chapter IV, which discusses the GUI replacement.

3.2.1 AAARF in General

3.2.1.1 Configuration Control By today's software system size standards, AAARF is not really a "large" system. However, when considered in the context of being the responsibility of

a single individual, AAARF is indeed large. There are currently twenty directories directly below the main AAARF directory, about half of which contain AAARF common code. The remainder contain algorithm class code and examples. The entire system, compiled and linked occupies about 25 megabytes of disk space (including the source files). The recommended method for using AAARF is for users to each have their own copy.[1] There are typically around 60 GCS/GCE students at any given time; allowing each to have their own copy then uses about 1.5 gigabytes of disk space. Obviously, this is unacceptable.

Currently, AAARF is run out of the current thesis student's directory. The danger with this practice is that student directories are archived when students leave. These directories can be restored, but there are likely to be path or OS version problems, which can result in a great deal of effort to get the software running again. It is exactly this practice which resulted in the current situation of having multiple copies of AAARF on different file servers, none of which work properly.

Originally, AAARF was intended to be a single copy, multiple-user system. With the addition of the parallel animation facilities, it became "easier" to treat AAARF as a multi-copy, multi-user system. At the time this was not a problem because of the scarcity of workstations. Now that large numbers of workstations are widely available, it is no longer practical for each user to have their own copy of AAARF. Aside from the obvious disk space issue, configuration control is simply not possible in such an environment because the current AAARF thesis student is responsible for AAARF and not the system administrator (as would normally be the case). This situation is further complicated by the fact that AAARF should be resident on various file servers to satisfy the needs of the general student body. Currently, this alone requires five copies be maintain on various file servers throughout AFIT.

[1]This conclusion is somewhat ambiguous. There are no specific references regarding where to install AAARF in the user's manual or any of the three theses I reviewed. The only reference I could find was in the distribution README file which accompanies AAARF, which states "... In order to compile AAARF, you should do the following: 1. To make the job easier, the aaarf directory created by the tar tape should be in a user's home directory. ..."

AAARF is quite "complicated," especially for inexperienced users. For each user to have their own copy requires AAARF be compiled somewhere off the users home directory. The instructions for doing this are clear and concise, provided there are no problems during compilation. If problems occur, recovery for the novice user is nearly impossible. The *makefiles* for AAARF are very good and very extensive. They are also quite unintelligible to anyone unfamiliar with the Unix *make*[23] facility. If the user intends to use AAARF for parallel algorithm animation, they must also have the necessary data collection programs and libraries in a directory called *aaarf* off of their login directory on the Hypercube. If the user wishes to run the AAARF parallel classes for the Set Covering Problem [29:Chapter 5] or the Shell Sort [29:Appendix A], these programs must also be in their respective directories below the users login directory. Clearly it would be best if all this were not an issue for the average user.

3.2.2 Parallel Computer Performance Monitoring AAARF's ability to animate algorithms remotely over the network is responsible for some of it's most complicated and challenging problems. First time users of AAARF instrumenting their algorithms for iPSC/2 performance animation must be prepared for a difficult time. The documentation for instrumenting an algorithm is incomplete and contains errors. There are no "simple" examples to guide first time users, only complicated, rather large, programs such as the shell sort and the set covering problem. A number of the pitfalls are described in the Sections 3.2.2.2 – 3.2.2.7. Before proceeding, however, it is necessary to explain how AAARF is configured for remote animation.

3.2.2.1 Remote Configuration For remote animation, AAARF is divided into two systems, one which runs on the display workstation, and one which runs on the remote host. For the purposes of this discussion we are concerned only with AAARF common processes, e.g. networking and data collection processes (see Figure 3.1). On the workstation side, this is the process *PRASEBG* (networking) and on the remote host side, the processes *server* (networking) and *aaarf_clct* (data collection). In this configuration, AAARF has two modes of operation: automatic

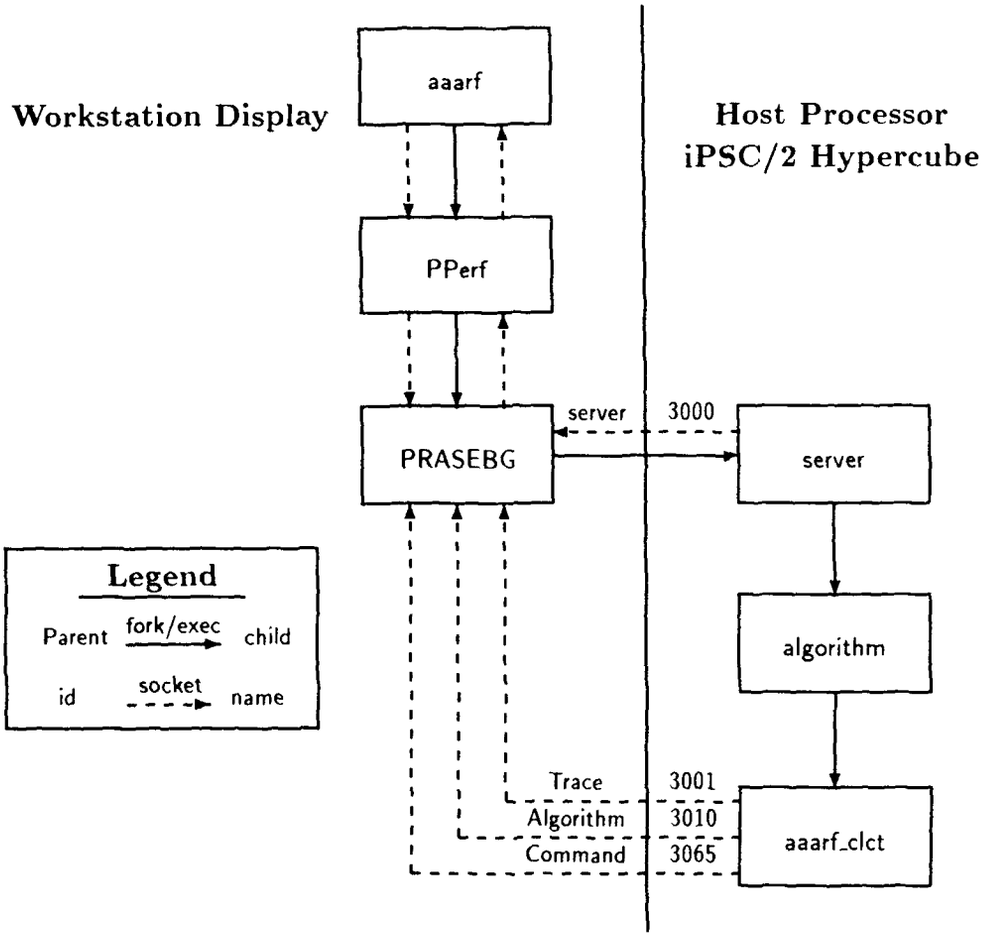


Figure 3.1. Network Connections for Automatic Mode. In manual mode, the *fork/exec* link between *server* and *algorithm* is not present. The algorithm must be started manually.

and manual. In automatic mode AAARF starts the algorithm being animated and in manual mode the user is responsible for starting the algorithm. In either mode, *PRASEBG* starts the program *server* on the remote host with the Unix *rsh* command. (*rsh* is the remote shell command. *rsh* connects to the specified hostname and executes the specified command[24].)

In automatic mode, the class being animated provides to *PRASEBG* the name of the algorithm to run on the remote host. This is in turn passed to the *server* process as a command line argument (via *rsh*), which in turn starts the algorithm. The algorithm being animated is required

to start the data collection program *aaarf_clct*. *aaarf_clct* then connects back over the network to *PRASEBG*.

In manual mode, no algorithm name is provided to the *server* process. It is the responsibility of the user to start the algorithm being animated as well as *aaarf_clct*.

3.2.2.2 User Requirements for Remote Connection As stated previously, AAARF uses the Unix command *rsh* to connect to the remote system. For this to function properly, users must have an account on the remote system and they must have a *.rhosts* file in their *login* directory [6:29]. Furthermore, they must define a *shell* variable, AAARF_SYSTEM, on the remote system and set it equal to the name of the workstation on which AAARF is running. The AAARF_SYSTEM variable is used by the remote data collection program *aaarf_clct* to determine with which workstation to connect. The problem with the AAARF_SYSTEM variable is obvious, it must be changed each time the user changes workstations.

3.2.2.3 The Notifier Environment AAARF uses a blocking read on it's sockets. The advantage with the blocking read is that it allows for very tight synchronization between communicating processes, which in turn greatly simplifies programming. The disadvantage is that it makes error recovery correspondingly more difficult. The reason for this is somewhat subtle.

Most SunView window events cannot be caught without using the Notifier (the Notifier is basicly a software interrupt handler [26]). The disadvantage with this is that once the Notifier is incorporated into the software, *all* software interrupts should be channeled through the Notifier[10:457–468]. This is accomplished by having the Notifier provide wrappers to the standard Unix or C functions which would generate a software interrupt. Consequently, the Notifier functions as a filter for all software interrupts directed at a process running under it's control, regardless of their origin (window system server, window manager, keyboard, timers, pipes, sockets, signals, etc.). The obvious advantage in using the Notifier is that it provides centralized control.

What all this means is that the default interrupt handling associated with a process is removed, e.g. ^C no longer works when, for example, a process is blocked on a socket read. This leads to perhaps the most frustrating problem with the current version of AAARF. If a *read()* on the socket communicating with the background process is issued before the background process is fully connected to the remote machine, the entire system freezes [6:30]. When this occurs it is usually necessary to login remotely and kill all AAARF processes, both on the display workstation and on the iPSC/2.

3.2.2.4 Hard Coded Socket Ids For communications on the local workstation, AAARF allocates sockets dynamically before forking the child algorithm process (remember with Unix, child processes inherit the parents file descriptors [24]). This is not possible for remote animations because there is no way to guarantee that the corresponding socket identifier will be available on the remote host. Consequently, the socket identifiers for remote animation are hard coded. If the either the local workstation or the remote host is using these ids, then AAARF cannot be run in the remote animation mode. Changing these socket identifiers requires recompilation for both the affected AAARF source files and libraries and the remote data collection programs. Obviously, choosing socket ids which are not likely to be in use is difficult at best.

3.2.2.5 Configuration Fragility A typical AAARF inter-task communication network for remote animation is shown in Figure 3.1. This configuration is very fragile because any break in the link may cause the display screen to lock up. Once locked, users must login remotely to kill the offending process(es). The problem with this is very simple, but not at all obvious to the novice user: the only process named AAARF is the main AAARF process. Every other process has a name related to either the class being run, or it's respective function within AAARF. If AAARF is killed, some of the background processes remain alive, holding the sockets as resources and preventing AAARF, or at least that algorithm class, from being run again.

3.2.2.6 Detecting Remote Problems Detecting problems (under program control) once the *server* program is started on the remote host is best described as difficult. Typical problems include incorrect paths for the class program to execute, and problems detecting any kind of file system error (such as compressed data files). While these kinds of errors are usually detected by the remote algorithm, AAARF has no facilities for passing this kind of information back over the network to the workstation. Unix provides a pseudo facility as a byproduct of the *rsh* and *fork/exec* facilites. *forked* processes inherit their parents file descriptors, which means that any *printf()* statements executed by the *server* or the remote algorithm (assuming automatic mode) should appear on the workstation in the window from which AAARF was launched. This is weak, but it works.

3.2.2.7 iPSC/2 Hypercube Node Program Termination An interesting problem which was discovered during this investigation involves iPSC/2 node termination. If the node programs are terminated with a *killcube()* command from the host process, rather than being allowed to terminate "naturally," the AAARF data collection process running on the host does not receive the information required to generate the end of data message for AAARF. As a result, the animation does not "finish" properly, and may in some cases result in abnormal termination of the animation process itself.

Another problem involving node termination occurs when node programs use the C *exit()* [12] function to terminate normal program execution. Although it is not documented in the users guide, the *exit()* function is redefined to be *praseexit()* by the instrumentation software. (*praseexit()* is the function which sends the end of data message to AAARF.) This is done in order to catch those situations for which the programmer has inserted a call to *exit()* to terminate processing abnormally due to some predefined error condition. The recommended method for "normal" node termination, as specified in the AAARF Programmer Guide [5:70-72], is to add the function call

praseend() (which also sends the end of data message) to the end of the node main process. If the author of the node program is using a call to *exit()* to terminate the node process, then adding the recommended call to *praseend()* results in multiple notifications of node termination being sent to the data collection process on the host. The result is a completely locked screen, once again requiring remote login to clear.

3.3 Recommended Changes

3.3.1 AAARF in General

3.3.1.1 Configuration Control For AAARF to become a generally available and useful software tool it must be treated more formally as an application program, with appropriate configuration control, and less as a thesis student project. This requires that *someone other than the current thesis student* be given responsibility for AAARF configuration management. Finding an individual for this task is a complicated problem in an environment where there are so many workstation clusters and file servers.

The default location for AAARF at AFIT should be the file server *olympus*. The AAARF version kept there should be the baseline version and no changes to this version should be allowed without the permission of the *olympus* system administrator and the AAARF faculty advisor. Availability of AAARF beyond *olympus* should be considered on a case by case basis with the goal of keeping the number of copies of AAARF to a manageable minimum. This is the only way to guarantee that a working version of AAARF is always available.

3.3.2 Parallel Computer Performance Monitoring

3.3.2.1 User Requirements for Remote Connection The problem of making AAARF a single-copy, multi-user system is primarily related to AAARF's remote animation facilities. It is the network connections which complicates the process of running AAARF to the point that it

has become easier for each user to have their own copy of AAARF (and the necessary files on the remote machine). The background process *PRASEBG* running on the display workstation uses the user's login path as the default path once the connection is made to the remote host. Obviously, any programs which are to be remotely started by AAARF must be in the users directory space. The most elegant solution to this problem is to obtain permission from the iPSC/2 system administrator to establish a directory specifically for the AAARF instrumentation and server code on the iPSC/2 and change the *PRASEBG* program to always look there when connecting. Class algorithms could also be placed in (or below) this directory.

At the same time, the requirement for the DISPLAY variable must be eliminated. A casual inspection of the problem suggests there are several possible alternatives (refer to Figure 3.1).

- The first is to change *PRASEBG* to add the name of the display to the command line argument list for the *rsh* call which remotely starts the *server* program. This would allow the *server* to in turn pass the display name on to the algorithm being animated, which in turn passes it on to *aaarf_clct*. The only drawback to this process is that it slightly changes the algorithm instrumentation process.
- The second case makes the same changes to *PRASEBG*. The server program, however, does not pass the display name to the algorithm being started. Instead, it declares a local environment variable called DISPLAY and sets it equal to the display name passed by *PRASEBG*. It then makes this variable visible to child processes using the Unix *export*[24] function. The advantage to this solution is that no changes to the instrumentation process are necessary.
- The last method is to again make the same changes to *PRASEBG* and have the *server* program write the display name to a scratch file in the /tmp directory. *aaarf_clct* then reads this file to get the name of the display workstation. As in the previous case, no changes to the instrumentation process are necessary.

A closer look at the inter-process relationships of the three processes on the iPSC/2 within the context of the two modes of operation (automatic and manual) reveals a problem not addressed by the first two options. In manual mode, the parent-child relationship does not exist between the *server* and the algorithm, consequently it would not be possible to pass the display name via command line arguments or an environment variable. Thus, the only viable alternative is the last.

3.3.2.2 The Notifier Environment AAARF's reliance on the Notifier should be removed completely. The reason for this is discussed more fully in Chapter IV, but focuses on making

AAARF platform independent. As long as AAARF is dependent upon features or facilities specific to the Sun platform, it is not possible to port AAARF to other architectures.

AAARF's dependence on the Notifier is a particularly vexing problem. Because the Notifier is an integral part of the SunView model, it is necessarily an integral part of AAARF. Removing AAARF's dependency on the Notifier will be a difficult and tricky task because the centralized control afforded by the Notifier will have to be split over several facilites, e.g., signal handlers, window managers, etc.

3.3.2.3 Hard Coded Socket Ids The problem of hard coded socket ids is one that is not likely to go away. Several alternatives have been explored and none appear any more reliable than the current method. One method which shows promise is to have PRASEBG dynamically allocate the necessary sockets, and then pass these as parameters to the *server*. The sequence of events then follows roughly that of passing the workstation display name (see Section 3.3.2.1). With this method there is at least a 50/50 chance that the corresponding id will be available on the remote host.

3.3.2.4 Configuration Fragility The problem of connection fragility arises from the synchronous (blocking) reads that AAARF uses for sockets. The obvious solution is to use asynchronous reads (this is done with the Unix *fcntl()* function with the O_NDELAY switch [24]). Unfortunately, changing a synchronous program into an asynchronous program is never easy, especially with a system as large as AAARF. The ideal approach is to identify those blocking reads which can potentially cause the system to lock up and replace them with asynchronous reads. This may limit the number of changes necessary and potentially simplify the process.

3.3.2.5 Detecting Remote Problems The only way this can be implemented is by changing the instrumentation system to allow for algorithm fault reporting. This can be done in one of two ways:

1. Use the Interesting Event (IE) facility. AAARF's IE facility is the basic unit of communication between the algorithm being animated and the animation process [5:Chapter 5]. This is a very attractive method because the infrastructure already exists. Since each class has its own animation process, it allows for tremendous flexibility. There is even room for standardization. The only real drawback is that it would not work in manual mode because IEs are not used in this mode.
2. Modify the PRASE system (*aaarf_clct*) to include the reporting of application errors. Currently, only the context of node communication is monitored, not the content. This could be changed to allow the reporting of exceptions so that *aaarf_clct* could notify PRASEBG whenever serious errors were detected by the algorithm being animated.

The only viable alternative appears to be the second one. While it is not as elegant as the first alternative, it is more complete because it will work for both the automatic and manual modes of operation.

3.3.2.6 iPSC/2 Hypercube Node Program Termination The first priority with node program termination is to provide adequate documentation and examples of proper node program instrumentation. The best way to accomplish this is to provide directories containing examples of instrumented programs which have been thoroughly tested. Each should contain a README file with any caveats clearly explained. The programs chosen for animation should be representative of the applications typically run on the iPSC/2 and should include some very simple programs for programmers new to the iPSC/2 (such as the Intel supplied Ring demonstration program). At the same time, an expanded users manual with detailed explanations and examples should be developed. The users manual should mirror the example directories and should contain references to the programs in the directories.

It may also be possible to programmatically prevent the *exit()* problem discussed in Section 3.2.2.7. Careful examination of *aaarf_clct* reveals that the following code segment

```
{if((end_count == total_pids) && (iprobe(-1) == 0))}
```

is used to determine when to send the end-of-data message to AAARF. Simply changing the sense of the equality check between *end_cound* and *total_pids* may be sufficient, but thorough testing is required to ensure that other problems are not created.

3.4 Implementation and Analysis of Results

The focus of the operational maintenance phase is to make AAARF runable, and, time permitting, make it easier to use and more reliable. The following sections describe the results of implementing the recommendations of Section 3.3. Again, only those recommendations which were deemed necessary to satisfy the stability and benchmark requirements were implemented.

3.4.1 AAARF in General At the beginning of this investigation there were three copies of AAARF available, none of which ran properly. After some initial failures, one was eventually compiled and tested. It functioned properly, with the exception of one of the parallel performance animations. This turned out to be a simple error in one of the *include* files and was quickly fixed. The remaining systems were archived.

3.4.1.1 Configuration Control AAARF has been moved to the *olympus* file server, in */olympus4/aaarf*. Currently, only the AAARF thesis student and the *olympus* system administrator have write access to this directory. No changes to AAARF were necessary to accomplish this. A very positive effect of this change was to relieve the current AAARF thesis student of the burden of keeping AAARF running for other researchers while AAARF was in an unstable state during the GUI replacement.

At the same time, the corresponding AAARF code on the iPSC/2 was moved to */usr2/aaarf*. This did require some changes to AAARF. *PRASEBG* was changed to look in */usr2/aaarf* for the *server* program. Previously, *PRASEBG* defaulted to looking for an *aaarf* directory below the

user's home directory (*$HOME/aaarf*). The resulting AAARF system has been used throughout the Summer '92 quarter with no problems reported.

The issue of responsibility has not been settled. However, a better transition mechanism has been instituted by requiring the new AAARF thesis student to work with the current thesis student during the six month overlap period. The transition has been formalized by requiring the new student to take two hours of CSCE699 during the fall quarter and placing him under the supervision of the current thesis student.

So far, this has been very successful. Several new parallel algorithms have been instrumented and incorporated into the AAARF system as a direct result of this formalization.

3.4.2 Parallel Computer Performance Monitoring

3.4.2.1 User Requirements for Remote Connection The recommended method for eliminating the AAARF_SYSTEM environment variable (Section 3.3.2.1) has *PRASEBG* passing the display name to the *server* program as a command line argument to the *rsh* call. The *server* program then writes this name to a scratch file in */tmp* where it is subsequently read by *aaarf_clct*. (*aaarf_clct* also removes the scratch file upon termination.)

These changes have proved very useful. AAARF users (including the author) no longer complain about having to change an environment variable every time they change workstations. It has also simplified training of the new AAARF thesis student as well as other researchers using AAARF.

3.4.2.2 The Notifier Environment There is very little in the way of maintenance that could be done to the Sunview version of AAARF regarding the Notifier. However, some changes were possible using XView. These are covered in detail in Chapter IV.

3.4.2.3 Hard Coded Socket Ids This problem was not addressed beyond what has already been discussed.

3.4.2.4 Configuration Fragility Several attempts were made at solving this problem, with no success. (Refer to Figure 3.1, page 3-4 for this discussion. Recall that blocking reads cannot be interrupted while running under Notifier control.) The socket read which appears to be the culprit is in the function *getIE()*. *getIE()* is a class specific control function [5:52] provided by the client programmer. It's purpose is to request the next IE from *PRASEBG* and send it to the animation process (in this case *PPERF*). It is in the file *aaarf/PViews/Control.c*. The source code for *getIE()* is shown below. As can be seen, the *read()* can only return successfully if data is available, and the only way to satisfy this requirement is for the operating system to block the read action until this occurs, which is exactly what Unix does.

```
TRACE_DATA *getIE()
{
  TRACE_DATA *IEpacket;
 int IErequest = IE_REQUEST;

  IEpacket = (TRACE_DATA *)malloc(sizeof(TRACE_DATA));
  if(!IEpacket){
    perror("animate(0) Malloc problem"); algKill();
  }
  /**** ASK FOR THE NEXT EVENT ****/

  if(write(algSocket, &IErequest, sizeof(int)) < 0){
    perror("animate(1) TX IErequest"); algKill();
  }
  /**** Get the next event from the child process ****/

  if(read(algSocket, IEpacket, sizeof(TRACE_DATA)) < 0){
    printf("ALG: reading alg socket - no data available\n");
    perror("animate(2) RX TRACE_DATA"); algKill();
  }
  /*** SET IE TO ADDRESS OF THE INTERESTING EVENT PACKET ***/

  return(IEpacket);
}
```

Analysis of what happens if it is assumed that a read which returns with no data is not a fatal error shows that the function which calls *getIE()*, *animateTheAlgorithm* ignores an empty *IEpacket* and effectively does nothing. This is very encouraging because no changes in the control structure are required to accommodate the non-blocking read.

resetPChild() is the function which allocates the sockets used by *getIE()* (also in the *aaarf/PViews/Control.c.*) To change the read from blocking to non-blocking, it is necessary to make a call to *fcntl()* with the parameters shown below in the code segment from *resetPChild.*

```
    /****** Start child process if first time or child has died ******/
    if(algSocket == -1 || write(algSocket, &resetCommand, sizeof(int)) < 0)
    {
     printf("\n\nStarting BG process\n");
     if(algSocket != -1)
           (void)close(algSocket);

     /**** SET UP SOCKETPAIR TO COMMUNICATE
           WITH THE BACKGROUND ALGORITHM ****/

     if(socketpair(AF_UNIX, SOCK_STREAM, 0, socket) < 0){
           perror("Opening socketpair"); algKill();
     }
/*
     fcntl(socket[PARENT],F_SETFL,O_NDELAY);
*/
     /**** START THE BACKGROUND ALGORITHM PROCESS ****/

     if((bgProcess = fork()) == -1){
           perror("Can't fork() background process");
           userWarning(NULL, "Can't fork() background process");
           exit();
     }
     if(bgProcess == CHILD){ /* start a child process */
           (void) close(socket[PARENT]); /* child close the [1] socket */
           (void) sprintf(c_socket,"%d",socket[CHILD]);/* use the [0] socket */
           if(execl(BGFILE, BGFILE, c_socket, NULL) == 0){
     perror("Can't execl() background process");
     userWarning(NULL, "Can't execl() background process");
     exit();
          }
     }
     (void) close(socket[CHILD]);      /* parent closes the [0] socket */
     algSocket = socket[PARENT];      /* use the [1] socket */
    }
```

This change was made under the assumption that *PRASEBG* had no way of knowing whether or not a read was pending.

The results were inconclusive. Using the non-blocking read did in fact allow the animation process to be interrupted. It also solved the screen freeze problem caused by users attempting to start the animation before *PRASEBG* is fully connected to the remote system.

However, as is often the case in software maintenance, it created more problems than it solved. Algorithm resets are disabled after the first reset, and the animation never returns control because it never gets the end-of-data message. An attempt at analyzing *PRASEBG* to see if the problem might be at the write end of the socket threatened to consume the entire thesis effort! *PRASEBG* is an amazing program. The majority of the code in *PRASEBG* is in the *main()* routine, and it is nearly ten pages of code.[2] *PRASEBG* is an incredibly complicated program. It manages communication over four sockets, all of which are active simultaneously.

Another alternative which was considered, but not tested, was that of adding a signal handler for ^C through the Notifier. This probably would work but does nothing for the attempt to wean AAARF from it's dependence on the Notifier.

This is a very difficult problem. Given the complexity of the problem and the knowledge that the current system works, the decision was made to suspend work on improving the network connection until after the GUI replacement was finished.

3.4.2.5 Detecting Remote Problems Adding facilities to detect remote problems requires major changes to the PRASE data collection system, *aaarf_clct* and AAARF. Because of the scope of making these changes the decision was made not to pursue this until after the GUI replacement was complete.

[2] On a whim, I printed the file on a line printer using standard 66 line, 132 character wide printer paper and measured the *main()* routine – it's nine feet long!

3.4.2.6 iPSC/2 Hypercube Node Program Termination Analysis of changing the inequality sense between *end_count* and *total_pids* in the code segment

```
{if((end_count == total_pids) && (iprobe(-1) == 0))}
```

from *aaarf_clct* shows that simply changing the sense of the inequality is insufficient. (*end_count* is the number of nodes which have reported a status of DONE. *iprobe()* returns the number of messages in the message buffer for that process.) The decision to send the end-of-data message to *PRASEBG* must be based on the assumption that the node programs will not finish in any particular order. If the user has incorrectly instrumented his/her program then multiple calls to *prase_exit()* are generated with the end result of over-incrementing *end_count* and placing extra messages in the process message buffer. However, there is no simple way to determine whether or not these remaining messages are node done messages, thus they must all be read. Consequently, the sense of the inequality is irrelevent. Note that the compound conditional is necessary because there is no guarantee that there will be a message in the buffer when this statement executes.

Currently, the best solution to this problem is education and examples. Appendix B is an extremely detailed example of how to instrument a simple iPSC/2 program for animation with AAARF. This appendix will be included in the new User's Manual.

3.5 Summary

The changes made to AAARF have succeeded in making AAARF more stable. Several research students have used AAARF during the Summer '92 quarter for analysis of parallel programs on the iPSC/2 and all have reported favorable results. The next chapter describes the work done to replace the SunView GUI.

IV. X-AAARF - Design and Implementation

4.1 Introduction

This chapter is divided into two parts: the first describes in the detail the selection of the Graphical User interface (GUI) chosen for the X based version of AAARF; the second describes the design, implementation, and testing of the new interface, and the results of making these changes.

4.1.1 A Prototype X-AAARF Early in '91, Williams developed [30] a prototype X window based version of the SunView AAARF. The *Athena* widget set was used and development was done on a Silicon Graphics Iris 4D. The prototype included the AAARF main process, the common library, and the array sort class. The prototype was written using an early version of the *Athena* widget set. This set was reported by Williams to be weak and very buggy [30].

Late in '91, Lack [13:Chapter 5] attempted to extend the prototype X-AAARF. Lack ported the prototype to a Sun 3 workstation and succeeded, after a great deal of difficulty, in compiling the prototype. Eventually, Lack was able to make several extensions to the prototype, but the overall system was fragile and prone to failure.

An attempt was made to evaluate the prototype X-AAARF in May of '92. By this time the operating system and window environment had been upgraded and the prototype would not compile. Concurrent with this, an analysis of X Window development environments was underway. Based on the results of the analysis (described later), combined with the inability to compile the prototype, and the knowledge that the prototype would probably require redesign and reimplementation anyway, the decision was made to abandon the prototype code and begin again.[1] Besides, it's a software engineering axiom that developers should not attempt to make deliverable products out of prototypes.

[1]The prototype X-AAARF was never intended to be anything more than a proof of concept demonstration, consequently Williams did not document his work. The documentation provided by Lack was too general in nature to be of much use in resurrecting the prototype.

4.1.2 SunView in an OpenWindows Environment Most contemporary students at AFIT are using OpenWindows simply because all of the Sun workstations currently in use at AFIT default to OpenWindows. Few students have any exposure to, or experience with, SunView beyond applications such as the LaTeX support programs like *dvipage* or *texsun*. Most are not even aware that the windows put up by these applications are SunView windows.

The real issue here is that SunView and OpenWindows window managers are incompatible [27:219]. The SunView and OpenWindows window managers each manage their own window stack. Thus it is not possible, for example, to put an OpenWindows window on top of a SunView window.[2] The end result of this is that applications running under the two window managers must share the screen in a tiled (non-overlapping) manner if access to both is required (this is, of course, the users responsibility). Orphaned Sunview windows are particularly onerous, usually requiring the user to logout and log back in to clear the screen. In extreme cases it may be necessary to re-boot the workstation. The color maps used by the two window managers are different, causing annoying flashes and changes in color [27:222] when the mouse is moved back and forth between windows owned by the different window managers. The window interfaces are also quite different between SunView and OpenWindows.

The issues described in the preceding paragraph are generally not a problem for the average user because they are usually running an application like *dvipage*. Such applications are of a type which are run and then quit with little or no interaction with other windows which might be on the screen. This is not always the case with AAARF. When animating remote processes it is necessary to have a window open to the remote host. There isn't always room on the screen for multiple non-overlapped windows and it becomes problematic when trying to run AAARF and control the remote process at the same time. The most infuriating problem occurs when an orphaned SunView window (which can't be closed or iconified) is covering the center of the screen and the user wants

[2] The reason for this is not at all obvious: the root *window* is managed by OpenWindows rather than by the SunView applications; the SunView applications effectively update the display without reference to the OpenWindows windows[27:219-220].

to log out because this is where the OpenWindows exit confirmation notice appears, and it can't be seen because it cannot be placed in front of the SunView window!

Other problems resulting from the incompatibilities are discussed in [27:Appendix B]. Appendix B of [27] also discusses actions users can take to remedy or lessen the impact of these problems but they are too complicated to be practical in the context of AAARF. The goal is to make AAARF easier to use, not more complicated.

4.2 Selecting a Replacement Graphical User Interface (GUI) - XView

This section presents the analysis which led to the selection of XView as the GUI for the X Window version of AAARF.

4.2.1 Analysis of X Window Development Enviornments Each of the candidate GUIs is examined within the context of the considerations itemized in Section 2.5.1, *GUI Requirements* (which the reader may wish to review on page 2-15) before continuing. As it turns out, two of these considerations, Target User Group, and User Friendliness (End User), are not issues specific to each GUI. These two are treated in a general manner before each of the GUIs are discussed.

The target user group at AFIT, obviously, is the AFIT student body, and in particular, students in GCS/GCE programs. With the large influx of Sun SPARCstations, the target user group has ceased to be the issue it once was. With the exception of students in the Graphics sequence, most students use SPARCstations for their research and day to day computing activities. It is assumed that the situation is similar at other universities and research/development organizations.

At the same time, as graphical user interface languages have evolved, the differences between them have slowly diminished to the point that most window environments are essentially the same. From a "user friendliness" perspective, the only real issues that remain are those which affect the programmers using these languages.

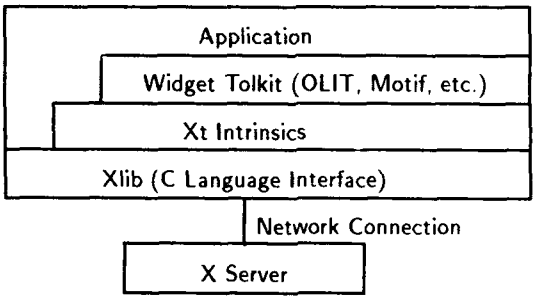


Figure 4.1. Programmer view of the complete X Window System [31:12].

Recall that each specification or standard is composed of two parts, the toolkit, and the environment. It is difficult to discuss one without also discussing the other. For this reason, the discussions which follow may at times appear confusing. Keep in mind that while the toolkit is the focus of the evaluation, the associated environment is necessarily included as well. (Figure 4.1 gives a visual context to the relationships between the various toolkits and the X Window System.)

4.2.1.1 **OLIT** OLIT is the OPENLOOK Intrinsics Toolkit and is based on the Xt Intrinsics toolkit. OLIT is one of two standards currently competing for the title of "Industry Standard" (the other is Motif). OpenWindows is the associated window environment. The OpenWindows environment is largely implemented with the OPEN LOOK toolkit (parts of it are still in XView) and it comes bundled with the OLIT development libraries. OpenWindows is the default user interface environment for the SPARCstation2.[3] It is running on all SPARCstation2s at AFIT. OpenWindows is the environment new students are first introduced to, and for many, the only one they ever use during their stay at AFIT (outside of PCs, which don't count anyway).

- *Compatibility* Not an issue.
- *User Friendliness - Programmer* As a "top level" widget set, OLIT provides all of the benefits normally associated with an object oriented paradigm. Within the context of AAARF, which has no "unusual" interface requirements, OLIT is completely satisfactory. It is also completely compatible with the C programming language. Oddly, Sun does not ship any OLIT development documentation, this must be purchased separately.

[3]This is an installation configuration decision. Sun bundles OpenWindows with the operating system. Of course, system administrators have the option of installing other environments.

- *Reliability* Reliability appears to be good. Admittedly, this is difficult to gauge. To help with software problems, Sun provides a known bug list with each release in $OPENWINHOME/share [22:4]. If workarounds are available, these are also included.
- *Portability* OLIT applications can be run under any window manager that is OPENLOOK compliant. Most major workstation vendors, and many third party vendors as well, supply OPENLOOK compliant window managers.
- *Enhanced Capabilities* OLIT does provide some extended capabilities over SunView. Most of these are convenience oriented, things which might simplify programming, but would not add any additional functionality to AAARF. In short, there is nothing about OLIT that would allow a previously unobtainable feature to be added to AAARF.
- *Development Effort/Time* This too is hard to guage. Most of the functionality of SunView is directly available. The difficult part is translating the functionality of the Sun Notifier into X Windows calls. In truth, this is not as difficult as it might appear on the surface; the X Windows server and window manager programs already function in much the same manner as the Notifier. Thus, it is possible to intercept window quit commands, etc.
- *Availability* Sun ships new workstations with OpenWindows and OLIT, including the development libraries.
- *Procurement Effort* Not an issue.
- *Procurement Time* Not an issue.

4.2.1.2 **Motif** Motif is the Open Software Foundation's user interface environment and toolkit. Motif is also based on the Xt Intrinsics toolkit. The Motif widget system is configurable and extensible. Configurable means that Motif widgets are *designed* to be used as is, or in combination with other widgets to form new widgets. (This is just inheritance.) Many of the "standard" widgets available with Motif are actually formed this way.

Motif widgets (instances) are dynamically allocated at run time. Widget classes are statically allocated with initialization data and operations (methods), and are inherited by instances at run time. (The resemblance to C++ is no accident, the Motif toolkit is written in C++.) A Motif widget instance, then, is composed of a dynamically allocated structure containing attributes and operations specific to that instance, combined with a pointer to a static data structure containing attributes and operations for every widget of that class. If the above is a composite widget, then the above is applied to each constituent widget in the composite object.

Motif widgets implement many of the high level functions associated with GUIs and normally associated with the windows typically found in a windowing environment: menus, buttons,

scrollbars, frames, forms, dialogs, etc. These are then combined to implement common window environment functions. An example is the FileSelectionBox widget which combines many of the above functions to provide the functionality to pop up a window, from which users can select or enter a file name. All of the functionality needed to search the current directory (with a mask or filter, if desired), present what is found, allow the user to select an entry from those presented, change directory, or enter a file name from the keyboard is either built into the widget or inherited from others. Obviously, this is very powerful stuff! With such high level functionality built into objects, user interfaces can be designed and implemented quickly.

- *Compatibility* Motif widgets are intended to be used with the Motif window manager (mwm), but also function well under the OPENLOOK window manager (olwm). Several Motif applications are in use at AFIT, and several have been developed by researchers in the database sequence. None have reported any problems that can be directly attributed to Motif running under the OPENLOOK window manager.
- *User Friendliness – Programmer* There is a formidable learning curve with the Motif toolkit.
- *Reliability* Unknown, but assumed to be at least as good as OLIT.
- *Portability* Motif "appears" to be more widely available than OLIT. It has been adopted by Silicon Graphics and Hewlett-Packard as the standard for their windowing systems.
- *Enhanced Capabilities* The same applies to Motif as applied to OLIT.
- *Development Effort/Time* Motif is somewhat more difficult to work with than the other widget sets. It is my opinion that the development time using Motif would be longer than that for OLIT.
- *Availability* Motif is not generally available on the SPARCstations at AFIT. Several small clusters have licenses, but these are not available to the general student population.
- *Procurement Effort* As a minimum, Motif would have to be purchased and installed on all of the workstations on the *scgraph* and *olympus* file servers (about 50 copies). The license fee is $54 per workstation. A floating license is also available, but this would only be practical for the *olympus* file server.
- *Procurement Time* Unknown. Since there are a few copies available, arrangements could probably be made to use one of these machines for development purposes while permission is sought to purchase the necessary number of copies. This is a delicate situation to be in, however, because there is always the risk that procurement can't or won't cooperate.

4.2.1.3 **Athena** The Athena widget set was developed by MIT in response to requests from users that a user interface widget set be included with the X distribution. The original set was considered weak and buggy. MIT has continued to expand and improve Athena, but the emergence of the OLIT and Motif toolkits has largely overshadowed the Athena widget set. The general consensus seems to be that the Athena set is not likely to last.

- *Compatibility* Not an issue.
- *User Friendliness - Programmer* The Athena set is widely considered to be one of the more difficult to work with because it is a fairly low level widget set. This was confirmed by Williams [30].
- *Reliability* Early releases were reported to have low reliability. The general consensus now seems to be that it has improved, however, it seems likely that MIT will adopt either OLIT or Motif as the distribution widget set once it becomes clear which is being adopted as the standard by the user community.[4]
- *Portability* Since X comes bundled with the Athena widget set, it is easily the most portable of the lot.
- *Enhanced Capabilities* None.
- *Development Effort/Time* Williams reported considerable difficulty in using the Athena widget set on the prototype X-AAARF [30]. Williams was also of the opinion that the set was limited in comparison to Motif and OLIT. It was felt that this limitation would unnecessarily lengthen and complicate development.
- *Availability* Not an issue.
- *Procurement Effort* Not an issue.
- *Procurement Time* Not an issue.

4.2.1.4 **XView** XView is an OPENLOOK compliant widget set from Sun Microsystems. The motivation for the development of XView is straightforwardly stated in [10:11]:

> "Today there are several thousand SunView applications, and one of the aims of XView is to make it easy to bring those applications to the X Window System marketplace."

The XView programmer's model was largely derived from the SunView model. Much of the functionality of the SunView model is retained, but the "look and feel" have been updated to comply with the OPENLOOK standard. The XView widget set is not as sophisticated as that of Motif. There are no high level widgets, like FileSelectionBox, and the objects are not configurable. In fact, XView is rather simple and rigid in comparison. The initial learning curve is small; programmers can quickly develop simple, efficient interfaces with minimal effort. However, to develop something equivalent to the Motif FileSelectionBox is a formidable task indeed!

[4] The X consortium recognizes two types of "standards," exclusive and non-exclusive. Xlib is an example of an exclusive standard - it is the only C-language interface to the X protocol. Non-exclusive standards, such as the Xt Intrinsics or the Athena widget set are considered part of the X window system (and the distribution system), but the Consortium may recognize other similar interfaces as well. [31:11]

Clearly, XView's most attractive feature is it's similarity to SunView. Sun has committed to XView by making the toolkit freely available; it is now part of the MIT X distribution as well as Unix System V [10:12].

- *Compatibility* Not an issue.
- *User Friendliness - Programmer* Of all the toolkits reviewed, XView has the easiest programmer model to understand and is far and away the easiest to program with. A number of simple, but effective examples are provided with the OpenWindows development software and are the same examples used in [10]. Since the XView model was derived from the SunView model, very little effort is required in making the logical mapping from SunView objects to XView objects.
- *Reliability* Reliability for early releases was reported to be poor. However, Sun appears to be committed to XView and later releases (including Version 3) have fared much better. More is said about this issue in Section[implementation and results].
- *Portability* Questionable. XView is currently being shipped by MIT as part of the X distribution. Since the Sun Notifier is an integral part of the XView model, it is not clear what this means in terms of portability.
- *Enhanced Capabilities* None.
- *Development Effort/Time* Based upon its similarity to SunView arguably the shortest of the lot.
- *Availability* Not an issue.
- *Procurement Effort* Not an issue.
- *Procurement Time* Not an issue.

4.2.2 Motivations for Choosing XView Based upon an analysis of the GUI systems surveyed above, XView was chosen as the GUI repalcement toolkit. Rather than go through an exhaustive analysis of why none of the other candidate GUIs were chosen, this section simply presents the motivations for choosing XView.

A great many factors were considered in making this decision. The driving factor was time: it was necessary to have a toolkit which was immediately available, and preferably one which would be quick to learn and easy to apply. None of the GUIs surveyed fit this requirement better than XView.[5] XView was designed with SunView programmers in mind. The syntax is similar, and in many cases, exactly the same. The similarity between the XView and SunView programming models facilitates the top down replacement strategy and can potentially accelerate the process.

[5] This is difficult to know *a priori*; sometimes one just gets lucky and guesses correctly.

XView is OPENLOOK compliant, which means end users do not have to learn another interface. Procurement is not an issue. Experimentation with XView during evaluation revealed that SunView code compiles under XView. Apparently, a comprehensive set of XView wrappers allows SunView code to be compiled directly into XView objects.[6] The process isn't perfect, and there is some loss of functionality, but it does further enhance the preferred top down replacement strategy.

4.2.3 A Closer Look at XView Before proceeding, a closer look at XView is in order. XView is an object-oriented toolkit. Unlike Motif, XView provides no explicit facilities for composite objects (there is nothing inherit in XView to prevent users from implementing that kind of functionality, but it is not needed by AAARF). XView objects are extensible, and facilities are provided for that purpose, but it was not necessary to make use of this feature for AAARF. The advantages and disadvantages of using XView are summarized in list form:

- *Advantages*
  - Most of the SunView GUI look and feel maintained
  - Notifier still available
  - Simple GUI structure
  - Very easy to use
  - Good introductory documentation
  - Source code for text examples included with OpenWindows
  - SunView code compilable in XView with minimal loss of functionality
- *Disadvantages*
  - Reliance on the notifier
  - No high level widgets
  - Object functionality often not clearly defined

The appearance of the Notifier in both lists is not a contradiction. The Notifier is needed to simplify the GUI replacement: as long as the Notifier is available, it is not necessary to find alternate means of handling events. This is necessary in the near term. In the long term, for portability reasons, reliance on the Notifier must be eliminated.

[6] Wrappers are an outgrowth of the recent interest in software reusability. Wrappers allow application code to be reused by translating function calls to old routines into the format needed by new routines.

4.2.3.1 The XView Programmer's Model This section presents an overview of the XView Programmer's Model as documented in [10:Chapter 2]. XView provides the programmer with a predefined set of interface components which are intended to simplify applications development. XView is an object-oriented toolkit, but the *object-orientedness* of XView is limited. XView objects are opaque and XView does not support composite objects at the user level. Although most XView objects are themselves composite objects, from the user's perspective XView supports only one level of inheritance, essentially that of instantiation.

Creating and Manipulating Objects XView provides a very clean interface to it's object set. There are only six routines:

- **xv_init()** Establishes the connection to the server, initializes the Notifier, initializes the Defaults/Resource-Manager database, loads the Server Resource Manager database, and parses any generic toolkit command line options. Called once at the beginning of the program.
- **xv_create()** Creates an object. Every XView object is created with this function.
- **xv_destroy()** Reclaims the memory allocated to an object.
- **xv_find()** Conditional front end to xv_create(). Searches for and returns an object with the specified parameters. If none is found, the object is created.
- **xv_get()** Get the value of the specified attribute for the specified object.
- **xv_set()** Set the value of the specified attribute for the specified object.

Using these six routines, programmers can create and manipulate the entire XView object set.

Types of Objects There are eight basic object types in XView. Three of these, *Generic Objects*, *Windows*, and *Openwins* are core classes and are not directly instantiable by the user. The remaining five are discussed below. The basic window entity is the *frame*. All other windows are classified as *subwindows* and must be attached to frames.

- **Frames** A frame is the basic window object to which the programmer has access. There are two flavors, a base frame, and a pop-up frame. A base frame is a frame with no parent. All other frames are subframes, so a pop-up is any frame which is not the base frame.[7] Each application has one base frame. There are no (preset) limits on the number of subframes. A frame is characterized by a border, which is managed by the window manager, and an interior which is configured and managed by the programmer. The window manager controls resizing, iconification, de-iconification, refreshing, quitting, etc. All XView windows are framed.

[7] Confused? So was I. The only way I could straighten this out was to write a program and instantiate a few frames and see what happened.

- **Canvases** A *canvas subwindow* is the XView graphics window. It's size is independent of the owning frame. The entire drawing surface is called the *paint window* and the visible portion is the *view window*. Multiple, scrollable views of a canvas are allowed within a frame.
- **Text Windows** A *text subwindow* provides basic text editing facilities. This window is a specialization of the canvas subwindow with text editing capabilities added.
- **Menus** Menus are not actually XView windows, but they are bound to windows at display time. XView supports a full range of menu types and options such as pull-down, pop-up and pull-right. Menus can be *pinned* to allow them to stay on the screen after the selection is made.
- **Scrollbars** Scrollbars are interesting objects. They can exist independently, or be attached to subwindows. Scrollbars are windows (because they are visible) but they are usually thought of as properties of subwindows. Scrollbars do not manage the objects to which they are attached, it is the programmer's responsibility to make the screen updates associated with scrollbar actions.

An important feature of XView that is not a window is the *Panel*. Panels implement the OPEN-LOOK *control area*. Panels manage *panel items*, e.g. buttons, sliders, text fields, and other forms of inputting data. The motivation for panels is to provide a mechanism for propagating events, especially for objects which do not have a window associated them.[8] Panels are very important in XView. For example, an application frame with no subwindows attached cannot catch interior mouse events. Attaching a panel to the frame allows these interior events to be propagated.

Obviously, there is much more to XView than what has been presented here, but this is sufficient to give the reader the necessary background for the design and implementation discussion which follows.

4.3 GUI Replacement - Design, Implementation and Results

The goal of the replacement process is to maintain AAARF in a functional state. Because of the expected lengthy time required to replace the GUI, and because considerable time was spent strengthening AAARF's state at the beginning of this thesis cycle, *no changes to the structure of AAARF which might jeopardize this goal were attempted.* AAARF is too big, the time at AFIT

[8]For the server or window manager to be able to propagate an event, that event must be associated with screen real estate. Buttons have windows associated with them, but menus do not. Thus, if a button has a menu associated with it, it is the panels responsibility to ensure that the menu gets the button push event.

too short, and the learning curve too steep to attempt major design changes in concert with the GUI replacement. Furthermore, it is absolutely vital that AAARF remain completely functional between thesis cycles. Once the GUI replacement is completed, consideration can be given to design change opportunities which might arise as a result using a different GUI toolkit (see Section 5.2.3.

This section starts with a description of the motivation for choosing the replacement strategy ultimately adopted. Next follows a general discussion of several high level design issues. Then each of the major components of AAARF is presented in a general discussion format which includes design, implementation, and results. Finally, the entire GUI replacement process is summarized.

4.3.1 Replacement Strategy The preferred method for this effort is to proceed in either a top down or bottom up manner. The decision as to which is appropriate is driven by the design of AAARF and the tool chosen for the job. Ideally, these two factors should be compatible enough to permit a structured approach.

As shown in Figure 4.2, AAARF uses a modular design paradigm. Each class system is a self-contained, executable system, e.g. each *could* be run stand-alone.[9]

Each GUI standard has two parts: the toolkit specification and the environment specification. For example, OpenWindows is OPENLOOK compliant because it is written in OLIT (which is OPENLOOK toolkit compliant), and the open look window manager (olwm) is OPENLOOK environment compliant. This is important because *there is no guarantee that GUIs compliant in one environment can run, or run correctly, in another.* For example, windows developed using OLIT may not run in the Motif environment. If they do, they may not run properly, or features may be missing. SunView is outside of all this. SunView windows work regardless of the X environment on the workstation. Thus, the initially perceived liability of SunView's independence is

[9] This is not strictly true because each class is required to open a socket for communication with the main AAARF process. However, the only interaction the main AAARF process has with the class system is to send a *kill* message if the user quits AAARF, or to send *stop*, *go*, and *reset* messages when running under central control. Otherwise the class system is entirely self contained and receives all it's input through its own window interface.

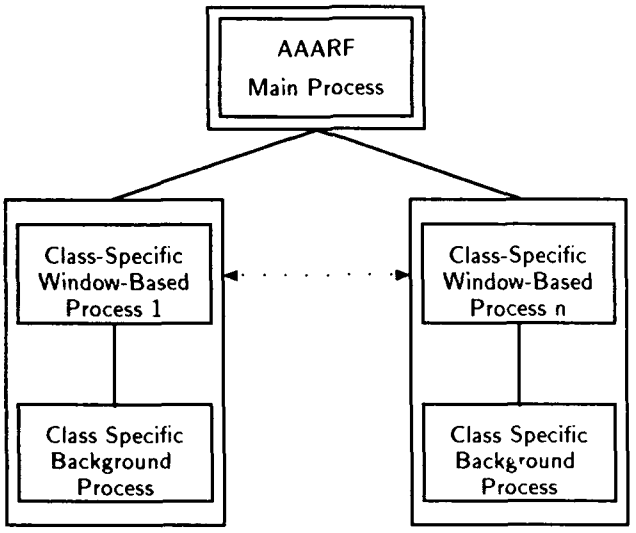


Figure 4.2. The AAARF modular design. Each outer box is a self contained system with it's own window based interface. Inner boxes are separate processes.

really an asset! The importance of this cannot be overstated! It means that while the AAARF main process' interface is under development, the class animations can still be run. Once the main process interface is completed and tested, each class can be done without affecting those which still contain the SunView interface. Clearly, the modular design of AAARF coupled with the ability to display SunView windows provides the ideal development environment for the replacement of AAARF's SunView GUI.

From a GUI perspective, the major components of AAARF are:

- *The AAARF main process.* The AAARF main process is the highest level interface to AAARF. It contains the main menu, the environment control facilities, and the central control facilities. Each has a window associated with it.
- *The AAARF common library.* Each AAARF class interfaces with the common library for animation control. There are three windows in the common library, the master control panel, the status panel, and the animation recorder.
- *The AAARF classes.* There are currently six AAARF classes. Each class has one window, the animation window, directly under it's control. Each class also "inherits" a copy of th common library windows at link time.

The goal is to selectively replace the GUI for each of the major components. Again, AAARF's modular design provides the necessary mechanism through it's use of the Unix *make* facilities[23:Chapter

5]. The main AAARF process, the class common library, and the six classes are organized as separate compilation units. A portion of the top level *Makefile* for AAARF is:

```
BASE = $(PWD)
LIBS = main common PViews
APPS = ArraySort PPerf Traverse Trees PSort PSCP
all:
        for d in $(LIBS) $(APPS); do (cd $$d; $(MAKE) BASE=$(BASE)) done
```

The arguments to *LIBS* are the AAARF libraries and the arguments to *APPS* are the six AAARF classes. Each of these arguments is a directory with it's own *Makefile*. By selectively removing these arguments, the *make* facility can be made to ignore them. The easiest way to do this is to move the unwanted arguments to the next line and comment them out. For example, to limit the compilation to just the AAARF main process, the *Makefile* looks like:

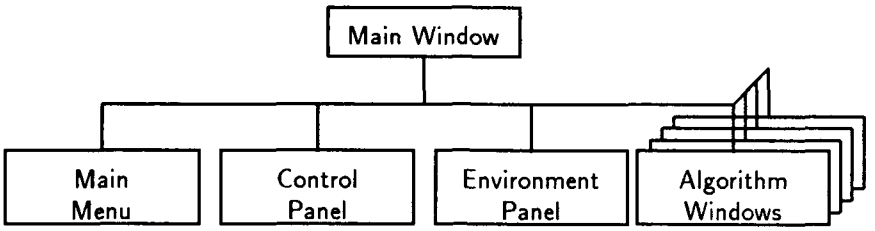
```
BASE = $(PWD)
LIBS = main
#common PViews
APPS =
#ArraySort PPerf Traverse Trees PSort PSCP
all:
        for d in $(LIBS) $(APPS); do (cd $$d; $(MAKE) BASE=$(BASE)) done
```

Thus, work can proceed on the AAARF main process, without affecting any of the executables built previously. Once the AAARF main process is completed, the class common library is done, followed by each class in turn.

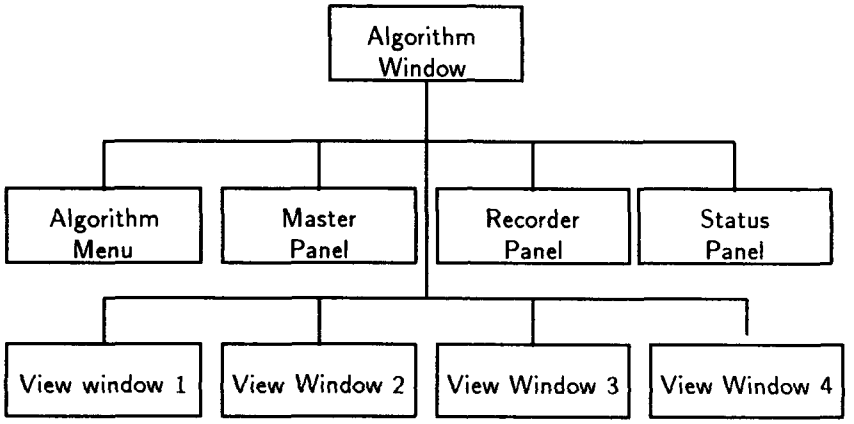
The net effect of all this is that AAARF can remain functional throughout the conversion process: SunView modules and XView modules can interact without interference. One could not ask for a "better" development evnironment! Based upon the above, a top down approach was chosen.

4.3.2 Test Strategy The general strategy is to test each module as it is converted. AAARF's modular design combined with the ability to display SunView windows naturally supports this. The preferred method for operational testing is to run each animation and exercise all of the Master Control Panel options. Each class should also be saved as an environment and restored, and recorded and replayed. Also, AAARF was given to the current AFIT CSCE586 class for use in a homework assignment involving the ArraySort class.

4.3.3 Design/Implementation Issues



(a)



(b)

Figure 4.3. (a) Original structure of the Main window. Each window is a separate window. (b) Original structure of the Algorithm window. The view windows are contained within the Algorithm window. The remaining windows are separate windows.

4.3.3.1 Algorithm Class Base Window The original AAARF window structures are shown in Figure 4.3. These figures show the SunView parent-child relationships between the windows. Note in Figure 4.3(b) that the number of view windows is limited to four.[10] These are not separately framed windows. Rather, each is attached to the Algorithm Frame as a SunView canvas subwindow, and each is dynamically resized depending upon how many are visible at any given time. The visual effect of this is to have the Algorithm Frame divided into (up to four) equal

[10] The motivations for the decision which led to limiting the number of views to four is not clear, however experience has shown that displaying four views causes serious performance degradation. A discussion regarding the motivation for choosing this relationship could not be found in [4]. The cause of the performance problem is unclear. The control structure which governs which views to paint (the class specific function *processIE()*) is simple and well structured and the associated graphics routines are short and uncomplicated. A more in depth analysis is required before the exact cause can be identified, but that is not part of this effort.

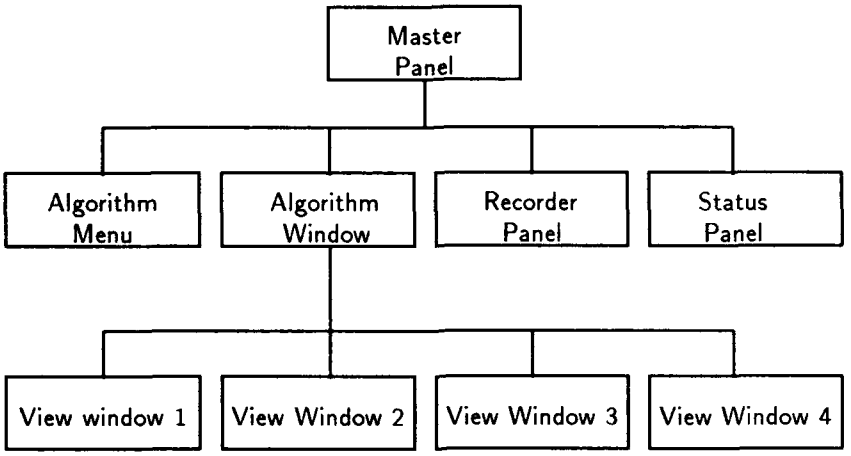


Figure 4.4. New structure of the Algorithm window.

sections, with one or more of the four visible based on user selection. The disadvantage to this is that the relationship between views in terms of size and and information conveyed is nonlinear, e.g for two views to convey equal levels of information may require the views be different sizes. With the current configuration, this is not possible. An alternative method allows each algorithm view to have it's own window. The advantage to this is that it allows the view window's size to be optimally adjusted for each view. The disadvantage is that it requires changing the core data structures and routines AAARF uses for window control. This design change was not implemented becase of the requirement not to make any core design changes during GUI replacement. However, changes in the relationship between the Algorithm window and it's Master Control Panel were made to accomodate incorporation of this change in the future. These changes in the relationship between the Algorithm window and it's Master Control Panel are described next.

In SunView AAARF, the Algorithm window is the base window for the algorithm class. The remaining windows in Figure 4.3(b) are children of the Algorithm window. Obviously, if each view is to have it's own window a new base window must be chosen. The Master Control Panel is the obvious choice. The Master Control Panel provides for all input, view, and animation controls for each algorithm class. Making the Master Control Panel window the base window requires changes

to the basic event handling structure of the class common code. These changes are restricted to the files *aaarfCommon.c*, *aaarfMaster.c*, and *aaarfViews.c* in the *common* directory. Figure 4.4 shows the revised Algorithm window relationships.

The decision to proceed with this change was the only significant design change made during the replacement of the GUI.

4.3.3.2 Graphics, Fonts, and Button Icons XView has a set of graphics functions, but their use is discouraged. Instead, the manual recommends the Xlib graphics routines. At the same time, XView also provides a set of wrappers for the SunView Pixwin graphics routines. The Pixwin graphics library is a small set of very high level routines, while the Xlib graphics library is low level. Concern was expressed that replacing the Pixwin routines might be very time intensive and prone to error so the decision was made to postpone replacing the graphics until after the GUI replacement was completed.

Fonts in SunView are not compatible with fonts in XView. For most of AAARF this is not a problem, the default fonts suffice. However, several of the animations are dynamically resized as the size of the search space changes. The graphics routines for these animations have font tables which they use to match the size of the font displayed to the size of the animation. The value of this capability is minimal since these animations grow very quickly. For this reason the dynamic font sizing was not carried forward into the new version of AAARF.[11]

The SunView AAARF Environment control panel and the Recorder panel both contain button icons. The icons were eliminated from the new version of AAARF in an attempt to reduce the size of the these two windows. The size savings amounted to about 30% for the Environment control panel and slightly less for the Recorder panel.

[11] The code has been commented out and appropriately documented in the event that future AAARF researchers require this capability.

4.3.3.3 XView Imposed Changes Because of it's similarity to SunView, XView did not force any changes in the existing AAARF design. XViews compliance with the OPENLOOK specification and the fact that it is an X Window System toolkit forced several implementation changes. These changes deal primarily with event handling and are caused by:

- the elevation of frame event handling from the application program to the window manager.
- the distribution of window events from the frame to other subwindows. In SunView, the frame catches most events. In XView, event catching is spread across the many subwindows. The SunView model is simpler, but the XView model is more flexible.

In general, event handling in XView is more sophisticated than in SunView. This is really inherited from X Windows and not due to any particular feature of XView; all of the toolkits surveyed provide similar capabilities. Overall, the impact on AAARF is minimal.

4.3.3.4 Matters of Style and OPENLOOK Compliance Openwindows applications supplied by Sun typically do not have an explicit "quit" or "kill" button or menu item outside of the one provided by the application's base frame. In order to maintain consistency across applications, XView AAARF does not provide any "quit," "kill," "iconify," or "de-iconify" facilities for any of it's windows. These functions are provided explicitly in SunView AAARF (via menu picks), but have been removed in XView AAARF. XView uses *openwin* windows for frames and these frames come packaged with event handlers for these actions. XView provides the necessary Notifier and callback facilities to intercept these actions if the application requires it. XView AAARF makes liberal use of these capabilities. The goal, from a users perspective, is make AAARF indistinguishable from other Openwindows applications.

4.3.4 The AAARF Main Process The first module to be replaced was the AAARF main process. This is the *main* directory, containing the files *aaarf.c*, *aaarfControl.c*, *aaarfMenus.c*, *aaarfEnvironment.c*, *aaarfWindows.c*, and *aaarfUtilities.c*. These files provide the top level interface to AAARF. The replacement started with an intense training session, reading the text [10] and running and modifying the examples in the XView source demo directory. The intent was to

gain proficiency, and an understanding of the preferred use of the XView objects. This was followed by an analysis of the SunView AAARF source code for the AAARF main process. This was done by printing the source files and highlighting the code that was expected to require changing. Diagnostic messages were added to the SunView code to help in tracing events so they could be more easily mapped to the XView object system. The primary goal was to establish a procedure which could be used for the remainder of the modules.

The AAARF main process consists of:

- The AAARF main menu, a pull-down menu that starts a user selected class, pops up the environment control panel, pops up the central control panel, or puts up a help screen,
- The Central control panel, a button window that is used to simutaneously control multiple algorithms,
- The Environment control panel, a button and text entry window that allows the saving and restoration of the current AAARF window configuration.

Several changes to the AAARF main menu were required because frame control no longer lies with the application. SunView AAARF provided "iconify," "de-iconfy," and "kill" as options in the main menu. These were removed from the main menu and added to the main frame event handler (see Section 4.3.3.4).

4.3.5 The Common Library The common library is one of the two most complicated modules in the AAARF system. It consists of the following files: *aaarfCommon.c*, *aaarfMaster.c*, *aaarfRecorder.c*, and *aaarfViews.c*. The Master Control Panel, as discussed in Section 4.3.3.1, was elevated to base frame status and the Algorithm Frame was made a child of the Master Control Panel. These changes proved relatively straightforward, although there was some initial difficulty because of weaknesses in the event handling descriptions in the reference manual [10:Chapters 6,7]. The Algorithm window in SunView AAARF has a pull-down menu associated with it, much like the main process. This menu pops up the Master Control Panel, the Animation Recorder, the Status Panel, and has menu picks for "iconify," "de-iconfy," and "kill." In addition, in SunView AAARF, selecting a class from the main menu causes only the Algorithm window to be displayed.

To see the Master Control Panel requires a menu pick from the Algorithm window menu. Since the Algorithm window was no longer the base frame, these functions had to be placed elsewhere. The only alternative was the Master Control Panel. The Algorithm window menu was deleted and buttons were added to the Master Control Panel for poping up the Status Panel and the Animaiton Recorder (see Figure 4.6). A positive side effect of changing the base window to the Master Control Panel was that it necessitated showing both the Algorithm window and the Master Control Panel at main menu selection time. This is seen as a plus because with SunView AAARF the first action users normally take is to pop up the Master Control Panel.

Most of the effort in the common library was spent in developing an intuitive, easy to grasp layout of the Master Control Panel. The SunView AAARF panel is well organized and functional, but several new panel objects are available in XView which are not available in SunView. One in particular is the *non-exclusive panel choice item*. This panel object displays lists and allows users to select multiple items. For example, there are seven possible views for the ArraySort class. Users may choose to display up to four of these simultaneously. SunView AAARF implements this as a non-exclusive choice pull-down menu. To select four views requires three right click-and-drag menu pull-down operations (there is always one view). With XView, all seven are displayed as small checkable boxes with labels, much like small square buttons with text beside them. Views are selected/de-selected with a left mouse click. This type of interface object is very fast, and very intuitive. Improvements of this type in the user interface satisfy the end user "User Friendliness" requirement specificed in Section 2.5.1. The original Master Control Panel is shown in Figure 4.5 and the redesigned Master Control Panel is shown in Figure 4.6.

The AAARF design precludes fully testing the Master Control Panel without a compatible algorithm class. This is because the algorithm classes provide most of the Master Control Panel's functionality. The AAARF common library provides only a skeleton control panel with several common functions. Buttons, sliders, and menus picks for CONTROL, INPUT, ALGORITHM,

LAYOUT, and VIEW options are added to the Master Control Panel by classes at run time. The ArraySorts class was chosen as the next module to be converted because it provides the most comprehensive set of interface options.

4.3.6 The Array Sort Class Converting the ArraySort class was relatively straightforward. A great deal of time was consumed in experimenting with the various XView panel objects in an attempt to reduce the size of the Master Control Panel, but this turned out to be a futile effort. In the end, real estate savings were sacrificed in favor of ease of use. The Master Control Panel interface is very solid, but testing with the ArraySorts class uncovered a problem in the event handling structure.

The XView *canvas* object provides an event hook (in the form of an attribute), called the CANVAS_REPAINT_PROC for a user supplied repaint procedure. The recommended method for redrawing a canvas window is to provide the name of a redraw procedure for this attribute [10:91-98]. When the server or the window manager detects that the window requires redrawing (a resize operation, or the window is uncovered), this procedure is called. In SunView, this is not a problem because the *frame* receives the event; only one event is generated regardless of the number of canvases attached to the frame. In XView, the canvases themselves receive the event. Since AAARF allocates all four canvases at run time, each canvas receives the event, resulting in some very annoying flashing while the screen is repainted four times. A "workaround" was developed but it proved extremely difficult and time consuming.

The workaround involved adding special conditions to several of the event handling routines in the common library. It was largely an experimental process. The problem is compounded by the fact that the event network, e.g. which functions get which events in what order, is not well defined in [10]. To further complicate the issue, mulitple WIN_RESIZE notifications are generated whenever the base frame is moved or resized, which adds to the flashing problem [10:122]. This problem is a direct result of having four views assigned to one window. XView allows this, but

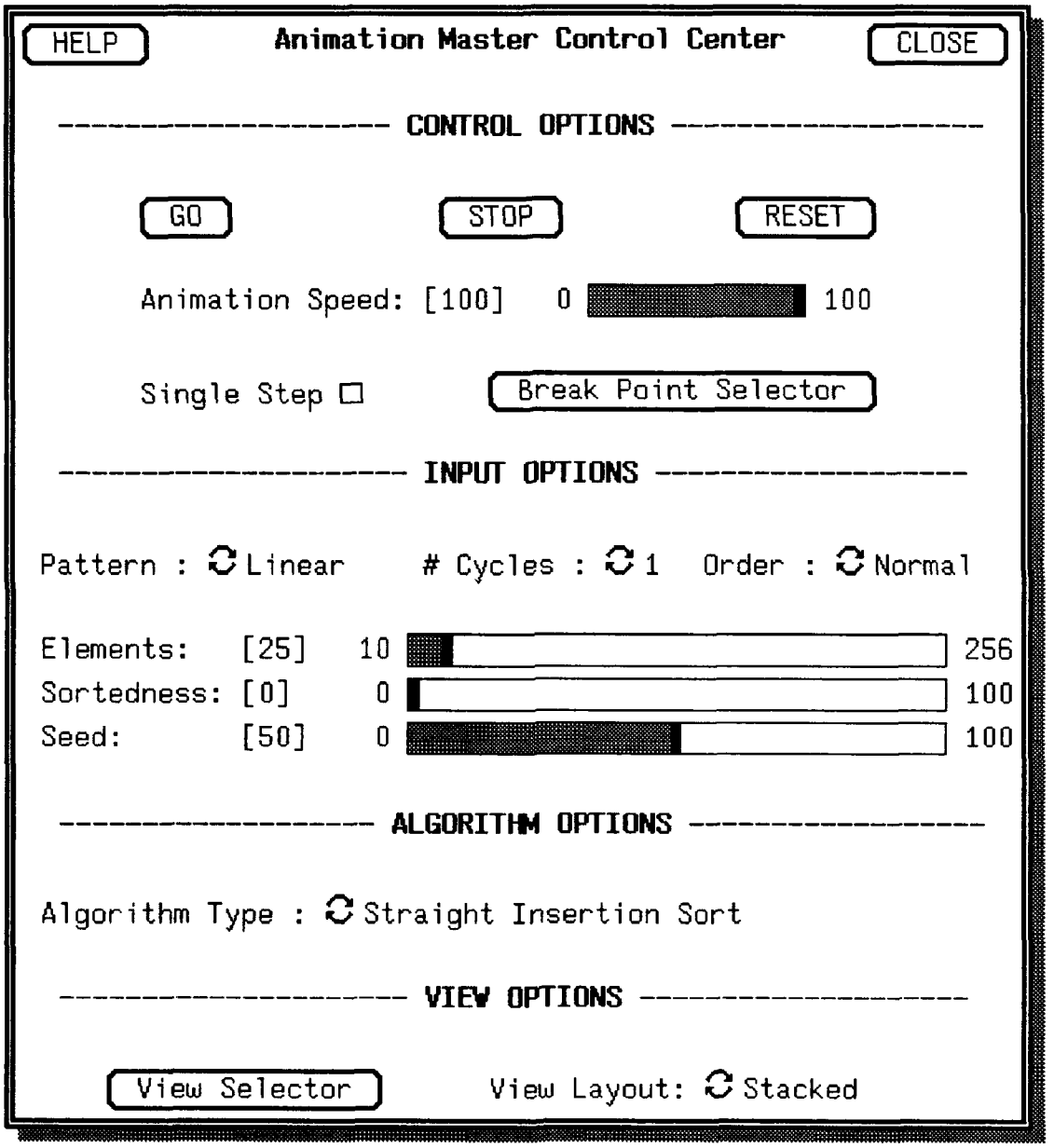


Figure 4.5. SunView Master Control Panel for the ArraySorts class.

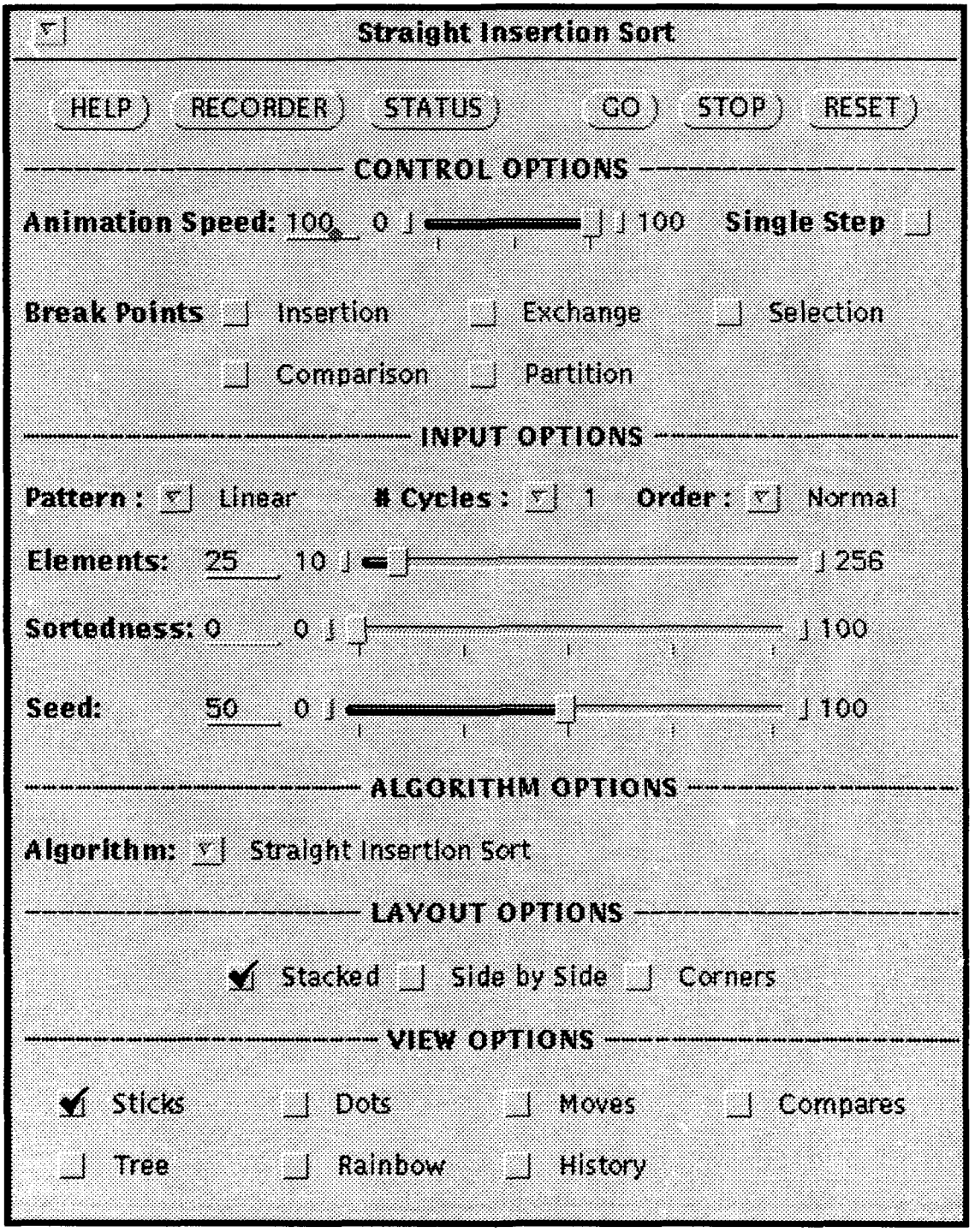


Figure 4.6. XView Master Control Panel for the ArraySorts class.

there is no discussion in the manual regarding the use of canvases in this manner. This is very likely non-standard use of canvases and it reinforces the opinion that the current method of having four views per window should be replaced with individually windowed views. If this is done, then the default event handling for canvases can be used, significantly simplifying the event handling.

Once the ArraySort class was completed, most of the "new" work replacing the GUI was done. The *Traversal* and *Traveling Salesman* classes were converted very quickly. That completed the serial animation classes. What remained was the parallel classes. The "other" of the two most complicated modules was next: the Parallel Views library.

4.3.7 The PViews Library The Master Control Panel for the parallel performance class is shown in Figure 4.7. The panel is not large enough to hold all of the control parameters necessary for parallel performance monitoring. As a result, an additional popup parameter panel was added by Williams in 1990 [29]. The panel is shown in Figure 4.8. The panel is activated by left clicking on the **Parallel View Options** button on the Master Control Panel. The work in converting the PViews library was relatively easy, but tedious because of the sheer volume of panel objects between the two panels.

The PViews library is designed to allow each parallel class to add algorithm specific panel items to the Master Control Panel and the Status display after the parallel performance objects have been added. In effect, this amounts to creating an object, and then updating it by adding list items. The SunView panel objects support this very "nicely," but XView does not. This problem occured in two places on the Master Control Panel, the **Break Point** and **VIEW OPTIONS** panel items. The changes resulting from this loss of functionality are the same for both panel items so only the **VIEW OPTIONS** item is discussed. (The discussion assumes the panel item has already been created in *aaarfMaster.c.*)

The algorithm class is required to provide the function *setViewItem()* to add the list of views to the panel object. A call to the PViews library function *setPViewItem()* gets the default parallel

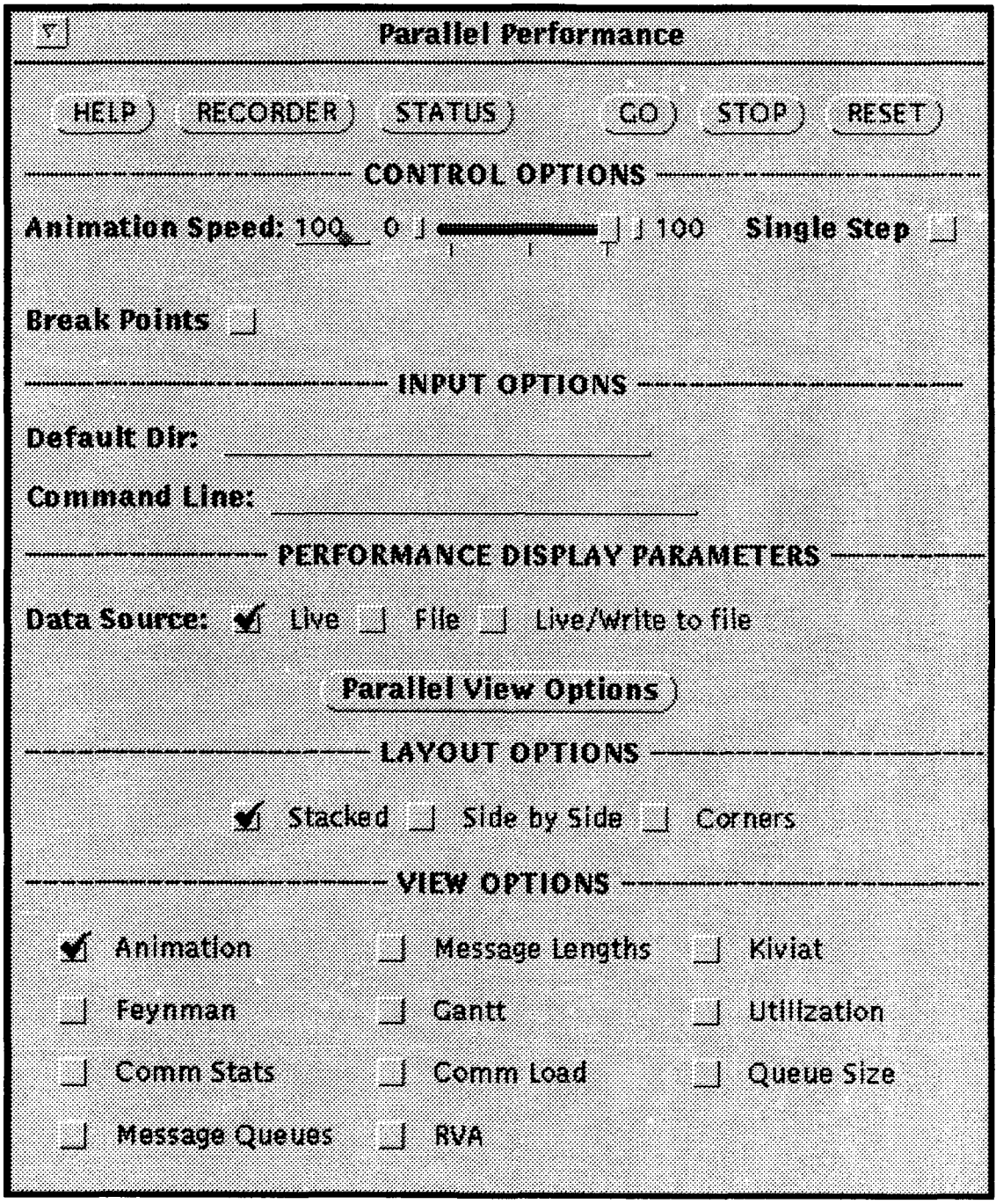


Figure 4.7. Master Control Panel for the Parallel Performance Class.

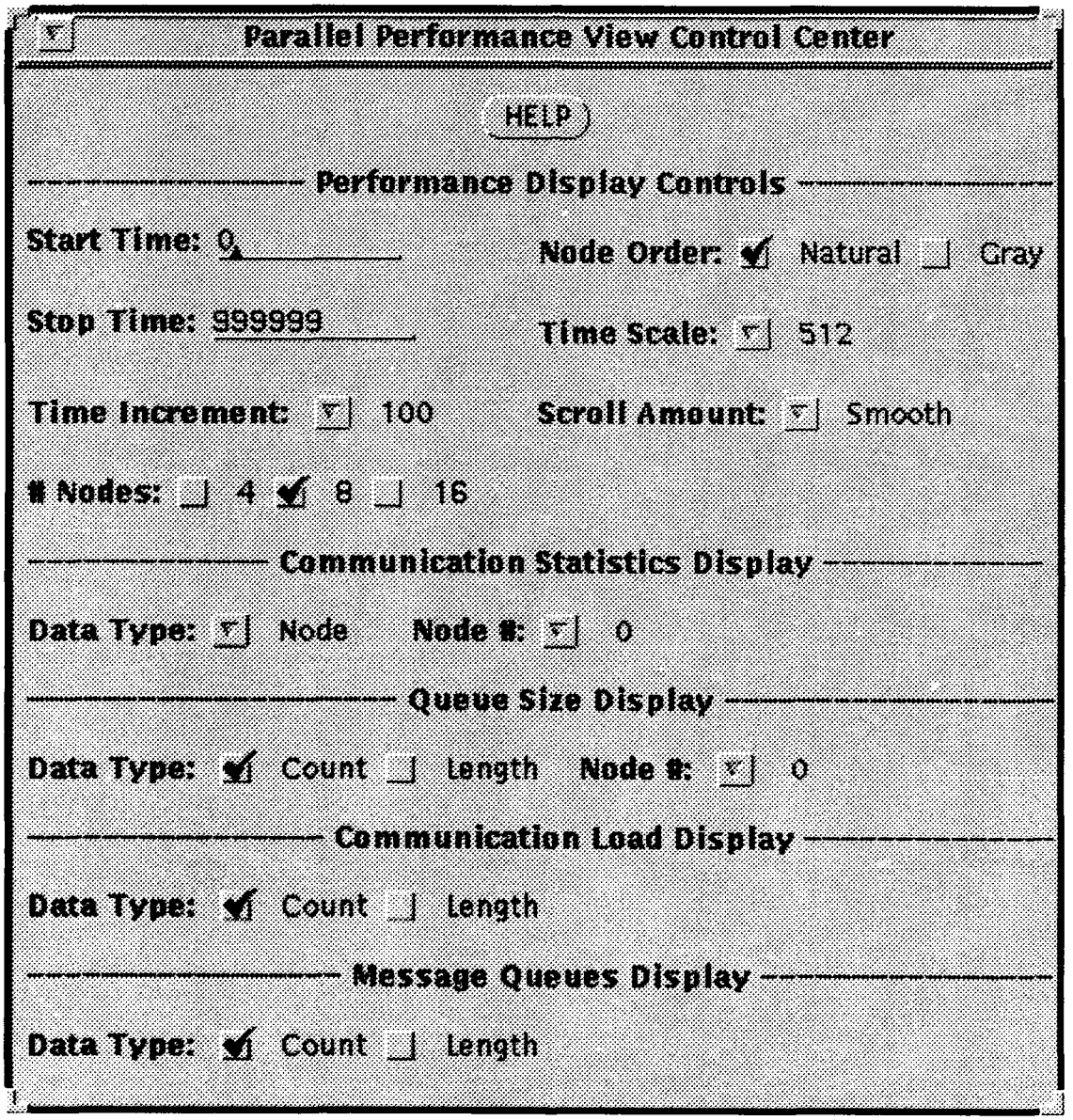


Figure 4.8. Parallel View Options panel.

performance views. This is followed by a *while* loop to add the class specific views. The original AARF code for setting the **VIEW OPTIONS** panel item is shown below:

```
void setViewItem(viewItem)
Panel_item viewItem;
{
  int index = 0, menu_index;
  static char *viewNames[] = {"Sticks",
                              "Dots",
                              "Rainbow",
                              "History",
                              0};
    /*
     *    set up the built-in views first, the number of other views varies
     */
    menu_index = setPViewItem(viewItem);
    /*
     * now set up the algorithm data views
     */
    while(viewNames[index])
    {
        panel_set(viewItem, PANEL_CHOICE_STRING,
                    menu_index+index, viewNames[index],
                    0);
        index++;
    }
    panel_set_value(viewItem, 1);
}
```

The XView panel object needed for the **VIEW OPTIONS** panel function does not support this. Consequently, the *setViewItem()* function call was changed to:

```
void setViewItem(viewItem)
Panel_item viewItem;
{
    xv_set(viewItem,
        PANEL_CHOICE_NCOLS,     3,
        PANEL_CHOICE_STRINGS,   /****  Default Parallel Views  ****/
                                "Animation",            /*      1 */
                                "Message Lengths",      /*      2 */
                                "Kiviat",               /*      4 */
                                "Feynman",              /*      8 */
                                "Gantt",                /*     16 */
                                "Utilization",          /*     32 */
                                "Comm Stats",           /*     64 */
                                "Comm Load",            /*    128 */
                                "Queue Size",           /*    256 */
                                "Message Queues",       /*    512 */
                                "RVA",                  /*   1024 */
                                /**** Algorithm Specific Views ****/
                                "Sticks",
                                "Dots",
                                "Rainbow",
                                "History",
                                "NULL,
        PANEL_VALUE, 1,
        NULL);
}
```

From a software engineering perspective, this is not a good solution. A better approach defines an array of strings in a header file containing the default views, with enough room for any ad-

ditional class specific views which might be added. This is then passed as an argument to the PANEL_CHOICE_STRINGS. It doesn't work. The data type for the PANEL_CHOICE_STRINGS is a "list" of character pointers (char *). Whatever a "list" of character pointers may be, it's definately not an array of strings. This problem is typical of those encountered with XView panel items.

Once the *PViews* library was completed, the three parallel classes, the *Parallel Peformance Views*, *Parallel Sort*, and the *Parallel Set Covering Problem* were converted. By this time, familiarity with XView and AAARF had reached the point that converting these three modules did not require any special effort.

4.3.8 General Results During testing a problem with the remote animation system was discovered. The background process PRASEBG's attempt to connect to the iPSC/2 caused two shell error messages to be printed on the workstation window from which AAARF was launched. The error messages indicated that a *stty* command option was illegal. AAARF does nothing with the *stty* command. The problem turned out to be in the *.cshrc* file on the iPSC/2. The file had two lines for setting the *erase* and *kill* values.

```
stty erase '^?'
stty kill  '^C'
```

These two entries had never been a problem when using the SunView AAARF, even when connecting from an Openwindows environment. But there is something about the XView version which causes these two lines to generate an error. The problem manifested itself in strange ways. Sometimes the animations would run to completion, and sometimes they would simply freeze the monitor. Occasionally, the animation window would just quit outright. The problem was "solved" by moving these two lines to the *.login* file.

The ArraySorts class was thoroughly tested by the AFIT CSCE586 class. Students were given a homework assignment requiring them to run the AAARF ArraySort class and evaluate it. It was decided to allow the students to use the new XView version. This was a difficult decision to make because work on the serial classes had just been completed, but very little testing had been done. Twenty-three students used AAARF over about a one week period. The results were very encouraging: no failures or errors were reported.

The parallel performance views have been thoroughly tested. Four parallel algorithms were instrumented and animated. Again, no problems were reported.

The process for converting modules developed during the conversion of the AAARF main process proved very successful. By the time the AAARF common library was completed, enough proficiency with XView had been garnered that the remaining modules were completed relatively easily. The value of having a conversion process cannot be over emphasized. It prevented oversight and helped immensely in partitioning the work to prevent overloading.

Finally, the new user interface has considerably simplified the use of AAARF. The XView interface has much the same look and feel as the Openwindows applications supplied by Sun. The annoying screen flashes of the SunView interface are gone, as is the necessity of having to remember two different interface protocols.

4.4 Summary

This chapter detailed the criteria used to select XView as the X-AAARF GUI. The design, implementation and results of implementing the GUI were presented. The resulting system has been thoroughly tested in accordance with Section 4.3.2 and shown to perform as well as the SunView version of AAARF.

V. Conclusions and Recommendations

This chapter presents conclusions regarding the longevity and durability of the new graphical user interface and suggests ideas for future work in this critical area. Recommendations for simplifying AAARF maintenance and recommendations for future algorithm animation research are also presented.

5.1 Conclusions

The XView GUI for AAARF has met or exceeded the requirements established in Section 2.5.1. As stated in Chapter IV's summary, the system has been thoroughly tested and the results are very encouraging. The look and feel of the Openwindows environment has been captured and AAARF's "ease of use" has improved accordingly. In the past, it was very difficult to perform maintenance on AAARF in the Openwindows environment because the SunView windows usurped the screen. With the new system, it is now possible to run AAARF while simultaneously analyzing source code files. For users familiar with the Openwindows environment, interacting with SunView windows has a feeling that is distinctly "unnatural." This was always present with SunView AAARF (and a constant source of comments like "Oh, well that's certainly obvious." drenched in sarcasm) but has completely disappeared with X-AAARF. Although they were vigorously sought, not a single negative comment specific to the user interface has been received.

The decision to use the XView toolkit appears to have been a good one. In general, the toolkit proved very easy to learn. The XView programmers guide [10] and reference manual [17] are well organized and complimentary. The reference manual was especially useful: once the basics of a particular object were understood, the reference manual usually sufficed.

For the time being, the XView interface is sufficient. It provides access to the many advantages of the X Window System and it is entirely adequate for the user interface needs of the AAARF system. It is not clear how long this will last, although Sun appears committed to long term support

of XView. If this continues, the XView interface may never need replacing. The state of AAARF is always of concern because it is by definition a system whose requirements are constantly changing – that is the nature and reality of research. The simplicity, elegance, and shallow learning curve of the XView toolkit are very desirable characteristics in such an environment.

The XView interface is not a solution to the larger problem of portability. XView is not widely available and it is not yet clear whether it ever will be. Greater availability of XView will not solve the portability problem unless the Sun Notifier system is included, or AAARF's dependence on the Notifier is somehow eliminated. This is not a simple problem – AAARF uses the Notifier for much more than window event handling. Indeed, much of the inter-task communication control hinges on the Notifier. The dependency on the Notifier for inter-task communication can be solved but only at the expense of a fairly large redesign effort.

In many ways, portability is a non-technical issue. For AAARF's intended uses at AFIT, portability is a not an issue. If it is desired that AAARF be distributed outside of AFIT, then portability is important. Distributing AAARF outside of AFIT is risky. There is no formal mechanism at AFIT for supporting, maintaining, and distributing software. The AAARF thesis student (assuming there is one) is the only individual capable of answering questions or solving problems that might arise as a result distributing AAARF, and it is a singularly bad idea to burden this individual with such a responsibility. AAARF could be distributed "as is" to interested users or researchers with the understanding that no formal support is available.

In reality, the problems of configuration control and maintenance are much more pressing issues. There is no AAARF "corporate memory" at AFIT. New thesis students are largely responsible for teaching themselves AAARF. As the size and complexity of AAARF grows, this task becomes more and more difficult.

5.2 Recommendations

5.2.1 AAARF Maintenance Without a doubt, the overriding difficulty in this investigation was in understanding AAARF. There is precious little detailed documentation. Hours were spent pouring over code and doing execution traces using diagnostic print statements. AAARF could seriously benefit from better documentation. The programmers manual provides a high level description of the client programmer interface with enough detail that users can implement a new class with a minimum of understanding of AAARF's control structure. There is no equivalent description of AAARF's control structure that would allow someone to make changes to the design of AAARF with the same "ease."

Currently, there are two manuals in the AAARF documentation set: the "AAARF User's Guide," and the "AAARF Programmer's Guide." A valuable addition to this set would be the "AAARF Maintenance Manual." This manual should contain as a minimum the following:

- A graphical representation of the event handling structures.
- A detailed description of AAARF's use of the Sun Notifier. This should be coupled with the event handling item.
- A flow chart of the control structure that is driven by the event handling structure.
- An annotated, graphical representation of the major data structures.
- A description of the inter-module relationships as they relate to the control structure.
- A graphical representation of the programmer interface discussed in the programmer's guide. This is not a client programmer issue, this information is necessary to fully understand the control structure.
- A listing of the Unix commands commonly used in maintenance and debugging. For example, the *grep* command is often used to locate the file containing a function or other item of interest.
- A detailed explanation of the AAARF *Makefile* facility.
- A flowchart of the network connection and management program PRASEBG. This will do as a minimum. The ultimate goal is to restructure and simplify the network interface.
- A section discussing the PRASE instrumentation software, the communications libraries, and the clock.
- An explicit example of PRASE instrumentation. (This has been done, see Appendix B.)

Some of the above information is available in the three AAARF theses. Some of it can be found in the user's and programmer's guides, and some as comments in the source code. What is lacking is a central repository for this critical information. A maintenance manual would solve this

problem, but this is a very complicated issue. This isn't research, so the question becomes, "Who should do it?" One possible answer is to have the AAARF thesis student(s) do it as a special study. The value in doing this is obvious: it would prepare the individuals involved for research involving AAARF and provide an improved training environment for future AAARF researchers. If AAARF continues to grow, and this problem is not addressed soon, AAARF's usefulness as a classroom aid and research platform will be endangered.

5.2.2 AAARF Training A more formal process of training AAARF researchers must be developed. The current process of encouraging students to "get inside the code" is no longer sufficient, AAARF is simply too large. This thesis cycle, the new AAARF thesis student was assigned to the current student as part of a special study. This proved effective in helping the current student pass along lessons learned, and many of the insights and tricks garnered during the course of doing research were passed along. There is little or no research value in this information. Instead, it is the kind of peripheral knowledge that never makes it into a thesis paper or documentation but is so vital to the success of a project. (Examples include knowledge of Unix and C, system quirks, shortcuts and tricks, LaTeX, etc.) This process can't really be formalized in an academic environment, but an informal process is better than no process.

5.2.3 Individually Windowed Views This is a very important requirement. The motivations for individual algorithm view windows were presented in Section 4.3.3.1. The individually windowed algorithm views should be implemented at the first available opportunity. The original AAARF design appears to have developed around the idea that an algorithm's animation is the central conceptual entity in describing an algorithm. In truth, it is only an artifact of the current state of the algorithm. From the point of view of AAARF, an algorithm's state is really the sum of all the window states associated with that algorithm, plus it's own internal state. Ideally, they are all consistent. In practice, they are rarely in sync because the animation and the algorithm are

separate processes. Which window is considered the "controlling" window in such an environment is immaterial. The decision is really an implementation issue.

To implement this recommendation requires changes to the parent-child relationships between the algorithm's windows and changes to the event handling and control structures which manipulate and track the state of the algorithm window and the algorithm itself. The best approach is to make the Master Control Panel the base window for each animation class, and all other windows subordinate. The necessary changes to the parent-child relationships between the Algorithm Window, the Master Control Panel, the Animation Recorder, and the Status panel were implemented as discussed in Section 4.3.5. Because of the requirement not to make major design changes during GUI replacement, no changes were made to the control or data structures. What remains is to modify the control structures which manipulate the algorithm VIEW_STATE data structure and the AAARF AAARF_STATE data structure and undo the event handling work-around described in Section 4.3.6. Before starting, a careful analysis of the impact of doing this should be done.

5.2.4 AAARF as a Classroom Tool - The Client Programmer Interface Currently, the learning curve for animating an algorithm as part of a classroom requirement puts it beyond the capability of most students. (A long term solution to this problem is discussed in Section 5.2.5.) Analysis of the problem indicates that the real issues are graphics programming and interfacing with AAARF. Students have the ability to design and implement algorithms of moderate complexity within a reasonable time (sorts, graph searches, traversals, etc). But, the additional task of developing animation graphics *and interfacing the algorithm and the graphics with AAARF* puts this effort beyond that which could reasonably be expected in a ten week course. There is so much to learn, in such a short amount of time, that the educational value would be jeopardized.

The near term solution is to develop a "student" animation class which already incorporates everything needed for animation except the algorithm, which the user supplies at run time. This requires the ability to do dynamic linking (such as that found in re-entrant OS routines and li-

braries). The functionality to simulate this is built into AAARF in the form of socket based IPC. AAARF simply treats algorithms as socket based "data generators" for it's animation programs. Conceptually, it's very simple, but ... . The suggested procedure for implementing this is:

1. Identify the specific algorithm classes (as defined in[4:11-14]) that are required to support the chosen curriculum.
2. Develop the necessary graphics routines for the chosen classes. This includes identifying the *Interesting Events (IEs)* needed to support the classes and their associated actions on the class data structures. It also means that the Master Control Panel and Status Panel functions to control execution be developed. Included in this is the requirement to prompt users for the name and location of their algorithm. This is the responsibility of the "student" class.
3. Provide to the users the IEs, their definitions, and a brief description of their use.
4. Provide a method for interfacing the user developed algorithms with AAARF. This must be as close to a "cookbook" recipe as possible, probably a *Makefile* in which users simply substitute the name of their program for a "dummy" name. This may well be the most difficult part of the process.

The perceived method for using the "student" class is:

1. start AAARF
2. from the AAARF main menu, select *New algorithm Window* --> **Student**
3. on the "student" Master Control Panel, click-select **Load Algorithm**
4. on the poped up menu, enter the path and name of the algorithm
5. once the algorithm is loaded, normal AAARF interaction applies

Users can then experiment with instrumenting their code and not have to concern themselves with writing graphics routines and interfacing with AAARF. This could be a very useful addition if the goal of using AAARF in the classroom is to focuses on algorithm behavior.

5.2.5 A Formal Specification Language for Algorithm Animation The process described in the preceding paragraph can be treated more formally, using the "student" class as a baseline. The *Interesting Event* concept can be used as a basis for the development of a formal specification language to describe animations in terms of graphics operations on visual representations of data structures. The expected benefits of such an hierarcical system include simplifying the process of animating algorithms, and increased flexibility and expressiveness in animating algorithms. The goal is to logically distill out of the animation process the necessity for users to write graphics

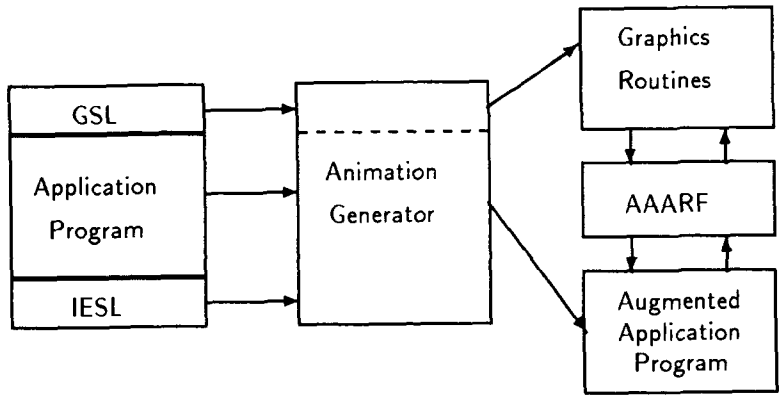


Figure 5.1. Proposed environment for the development of a formal specification language for algorithm animation.

routines, while still giving them the ability to specify and configure graphics entities. The ability to abstractly specify and configure graphics entities is missing from the solution recommended in the previous section. A prototype environment for the development and testing of a formal specification language for algorithm animation is shown in Figure 5.1 (This prototype is due to Bailor [1]).

Formalizing the *Interesting Event* concept is based on the hypothesis that each AAARF class can be treated as an application domain. A domain specific language can then be defined for each class. The domain language describes the data, operations (i.e., productions), and allowable states for the class. Two specification languages are necessary: the *Graphics Specification Language* (GSL) and the *Interesting Event Specification Language* (IESL). The GSL directs the mainpulation of graphic entities and the IESL describes the state of the algorithm. The GSL should be domain independent, while the IESL is domain dependent. An *Animation Generator* parses annotated programs and produces two outputs: the *Augmented Application Program* (the instrumented algorithm) and the *Graphics Routines* (the animation process). Users interact with these through the AAARF supplied class Master Control Panel.

If currently defined AAARF classes are treated as individual domains, then by definition, each class' set of IEs forms a basis set for that class, from which all possible states for that algorithm class can be formed. If this level of abstraction is too specific, then some subset of all the IEs

currently defined for AAARF can be used to describe broader classes. In any case, the IEs already defined for each class are a natural starting point in formulating a formal language.

Unfortunately, the SGL graphics operations are not so simple. Issues to consider are:

- *Level of Abstraction or Icons* Does the language manipulate points and lines, or higher level objects like polygons.
- *Static vs. Dynamic* How is movement defined?
- *Visibility* How is scale defined? Color?
- *Relationships* Can relationships between objects be formalized? How does this affect movement?
- *Construction* Can aggregates be formed? How does this affect movement and color? Visibility?

Currently, classes define their own graphics routines and register them with AAARF at link time. Classes do not have simple access to each others graphics routines. A better approach is to build into AAARF a basic graphic entity model and then allow users to choose, through the GSL, the entities and relationships which best fit their needs.

The discussion thus far assumes that it is the *user* who defines the relationships between the IESL and the GSL through the appropraite use of each language. Exactly how this is achieved is not clear. In one scenario, the user selects from the GSL the necessary productions to implement the desired graphics entities. Then, through some as yet undefined mapping process, the domain specific IESL productions operate on the objects output by the GSL productions to produce the desired effects on the screen. Various levels of observation are possible, depending upon the user and the algorithm.

The intent is to provide a formal mechanism for animating algorithms. An interesting side affect of such a system is that it is, in a sense, "reverse engineering." Given an algorithm, what formalisms are required to abstract the control to a higher level, e.g., a visual level where only qualitative relationships are observed? This is an interesting problem because it appears to have widespread application, not only in algorithm animation, but program visualization as well.

The worth of such a system is measured in it's ease of use. It serves no purpose to take an already difficult process and abstract it to a process at a higher level that is equally difficult. In this regard, the GSL is likely to be the most difficult to formalize because the two primary requirements for such a language – expressiveness and simplicity – are incompatible features. Effective use of the GSL depends upon users' understanding of the domain and the associated domain-specific algorithm animation environment.

5.2.6 AAARF Responsibility In a school which prides itself on it's Software Engineering program, it is an irony that no formal software Configuration Management program exists. This issue affects not only AAARF, but, to my knowledge, every other research program at AFIT involving follow on software development. Someone has to assume the responsibility for Configuration Management of AAARF. This is best done by a software engineer or analyst in a full time position.[1] The motivation is very simple: without an effective configuration management program a lot of research effort is either lost, literally, or wasted trying to restore previously functioning software.

5.2.7 The Future of AAARF In many ways AAARF has failed to live up to the expectations of it's designers. AAARF is too complicated for really insightful animation development as part of a classroom exercise, although AAARF is very useful as a pedagogical tool for studying algorithms at an "observational" level. If use of AAARF as an expanded educational tool is a goal, then the recommendations of Section 5.2.4 can be implemented to solve this problem. Failing to do so will likely result in continued non-use of AAARF for these purposes. This opinion is supported by the fact that no one outside of the two thesis students who developed AAARF have ever implemented a new class.

The use of AAARF as an analysis tool for parallel computer *performance* monitoring is a very delicate issue. There are other tools available, such as Paragraph, if the interest is solely

[1] I am keenly aware of the implications of this statement. If such an indivinal cannot be hired, then the only alterantive is the AAARF thesis advisor. Nevertheless, research software is a valuable asset and should be treated as such.

in performance monitoring (see Section 2.4.2.2, page2-9). Whether Paragraph is "better" than AAARF is a very complicated question. Certainly Paragraph is not as general in nature as AAARF: it does not support algorithm animation, and it does not support on-line performance monitoring.[2] However, AAARF's parallel performance animation repetoire is not as complete as Paragraph's. For performance monitoring, AAARF is entirely dependent upon the PRASE execution trace software. PRASE is non-portable; it runs only on the iPSC/2.[3] It is possible to adapt PRASE for use with other parallel computers, but it's not clear that this is desirable. Another alternative is to modify AAARF to accept PICL trace records (see Section 2.4.2.1, page 2-8). PICL is supported by ORNL and is becoming something of a trace format standard, and PICL supports a number of parallel computer platforms.

As pointed out in Section 1.2, page 1-4, AAARF is based on a very solid design. It is doubtful other performance animation systems share this heritage. With the above in mind, there are two possibilities:

1. Continue as is. The AAARF system is relatively stable and there is much that can be done in the arena of parallel performance monitoring. But, there are problems with this option:
   - The research content of much of this work is questionable.[4]
   - The size and complexity of AAARF limits what can reasonably be done because so much time and energy is expended in getting started. Researchers must first learn the basic AAARF system, then the remote animation facilities, and finally the remote instrumentation facilities. And, this assumes they are already familiar with Unix and C.
   - The AAARF system is in a constant state of flux, limiting its usefulness to other researchers. This is amplified by the lack of Configuration Management.
2. Abandon the parallel side of AAARF and concentrate on perfecting AAARF as an educational tool and a research platform for formal animation specifications. Adopting this option (potentially) sacrifices the ability to animate parallel algorithms. There are several advantages:

[2]Paragraph uses file input. AAARF is a direct connect or on-line system. Admittedly, there is *currently* no real advantage to on-line mode over file input mode because network bandwidths are too narrow to allow animations to run in any where near real time. This may not always be true. A long term goal for AAARF is to be able to run animations interactively, in real time. This capability does not currently exist in AAARF, but the basic functionality to support it is in place.

[3]PRASE is being ported to the iPSC/860, but this does nothing to enhance its portability. Intel is discontinuing support of the iPSC/2 at the end of calendar year '92, so PRASE is still, effectively, a single system platform. The two computers are so similar that only a recompile is required.

[4]There are always tradeoffs in an academic environment between what is research and what is development. This is a particular problem at AFIT because students are required to complete the program in 18 months, and extensions are rare.

- Adopting Paragraph/PICL allows researchers to concentrate on using these tools for analysis, rather than developing similar capabilities. The general trend in parallel computers is to incorporate performance monitoring instrumentation and animation facilities directly into the architecture and operating system. Some, like Intel, are using PICL and Paragraph. This knowledge weakens the motivation for continued in-house development.
- The potential for research in formal specifications for algorithm animation appears unlimited. This is a new and exciting area and there is much that AAARF's current configuration has to offer, starting with the recommendations in Section 5.2.4.
- The use of AAARF as an educational tool has great potential if the previous item is pursued. The system described in Section 5.2.4 could eventually be replaced by a more formal method based upon formal specifications.

Which path to choose is a difficult decision. Both would be ideal. Unfortunately, to date, there has only been one AAARF researcher per thesis cycle. To pursue both paths requires two people. In light of the current situation regarding downsizing of the Air Force, this is not likely to change. Much has been invested in AAARF's parallel capabilities, but largely at the sacrifice of AAARF's contribution as an educational tool. This is unfortunate because both are valuable research activities.

In the final analysis, the decision is not so much which avenue to pursue, it is more a question of where to focus a dwindling resource.

Appendix A. *A BRIEF Discussion of X*

X was (is) intended to be a distributed, device independent user interface platform. It's primary use is in the development of device independent (e.g. portable) Graphical User Interfaces (GUIs). X does not contain any particular user interface styles. Instead, it provides a set of device independent tools from which any number and kind of user interfaces can be built.

X is based on the client-server model. The core of the X system is the server. The server (display in X-ese) allocates and manages all the necessary data structures required to support a screen (output device in X-ese). There is one server per cpu, but a server can manage more than one screen (analogous to a file server with diskless clients). Applications programs using the server are known as clients. Any application which complies with the X protocol (an asynchronous byte-stream protocol) can communicate with the server. Obviously, a server can connect to many clients, but a client can also connect to more than one server. A client and server need not be on the same machine, or even the same network.

The server provides the device independent interface to the platform on which it resides. A specific version of the server must be installed for each platform. For example, in a networked workstation environment, each workstation has a device dependent server running in the background controlling the screen.

The lowest level of access to the server is directly through network packets and byte-streams. This would be analagous to microcoding. Not recommended. The next level of interface is through specific language libraries which provide a complete set of window management functions and capabilities. This is the assembly language level of X. XLib is the C language interface (there is only one per language). Above the library level are toolkits, such as the Xt Intrinsics (also in C). Xt is built on top of XLib. (There are a number of other Xt level interfaces to X Windows built on XLib.) The highest level interface is what is generally known as a set of widgets. Widgets are basically a collection of objects built using a toolkit and XLib which implement a set of user

interface functions. For example, menu buttons, file dialog boxes, scroll bars, etc. are typical examples of widgets. Motif is a widget set. In general, a toolkit such as Xt provides a set of server interface functions for managing windows and a set of widgets for user interface functions. While the typical application may use the top three levels (down to XLib), most of the work is done at the widget and toolkit levels. However, most programmers never actually deal directly with XLib or Xt objects because the attributes associated with these objects are available through the widget objects via inheritence.

Appendix B. *A Simple Example of PRASE Instrumentation - The Ring Program*

B.1 Introduction

This document presents a "simple" example of how to instrument an iPSC/2 program for animation with AAARF. The example program is the Intel supplied ring program. Read this entire document before attempting to run the ring animation. Don't be dismayed by the size of this example. This is Unix - it's supposed to be hard. If it was easy, it would be on a Macintosh.

This example is divided into four sections:

- OVERVIEW (B.2)
- Instructions for RUNNING THE ANIMATION (B.3)
- Instructions for BAILING YOURSELF OUT (B.4) when you lock up the system
- Instructions for INSTRUMENTING THE RING PROGRAM (B.5)
- SOURCE LISTINGS (B.6) for the *ring* program

Examples of other parallel programs which have been instrumented can be found in other directories off the */usr2/aaarfDEMOS* directory.

B.2 Overview

The iPSC/2 */usr2/aaarfDEMOS/ring* directory contains the C source and makefile for the instrumented ring example. The intent is to provide a simple example of how to instrument a hypercube program for animation with the AAARF Parallel Performance Views class. In this case there is no AAARF class from which to run the ring program automatically, it must be started manually on the cube. The directory should contain the following files:

- README
- host.c
- node.c
- makefile
- rhosts

The directory contains a modified version of the Intel *ring* demonstration program. The *ring* demo has been modified to acommodate AAARF instrumentation. Ring count reporting has been eliminated (just because, this is not necessary for AAARF instrumentation), and a terminate message type has been added since the original version has no facility for gracefully terminating the node program (which IS necessary for the AAARF data collection system to work properly!). The original version of the ring program has facilities for multiple runs, which has also been removed. The following description of the ring demonstration is taken from the Intel README file which accompanies the ring demonstration program.

> The host program loads the node program and prompts you for input as to the number of times to go around the ring and the length of the message you want to pass around the ring. Node 0 receives this information and sends a message of the desired length to the next node (1). As each subsequent node receives the message, it sends it onto the next node in the ring. After the desired number of rounds have been completed, it reports the time the message spent "circling" the cube. To exit this program, enter a negative number when prompted for the number of times to go around the ring.

B.3 Running the Animation

Before you can run AAARF and the ring program you must have accounts on at least one workstation cluster on which AAARF is resident, as well as the Intel iPSC/2 Hypercube. AAARF is currently available on *olympus* and *scgraph*. To run this example you need to (explained in detail below):

- copy the source to the cube and make executables, e.g. compile them,
- get the cube,
- start AAARF,
- start the host program.

This program works on any size cube greater than zero without modification. You must know what kind of math coprocessor your cube nodes have. Enter "make help" to determine the

appropriate make command to enter to build the correct executables. (This demo can also be run without AAARF instrumentation by editing the makefile - remove the -DPRASE option wherever it appears - and recompiling.) AAARF runs on your workstation, and the instrumented program runs on the cube. They communicate over the network via sockets. Because of this, you MUST follow very carefully the script described below. If you do not, you will likely lock up both your workstation and the cube. Instructions on how to dig yourself out of such a situation are provided later. Finally, AAARF prints certain information and diagnostic messages in the window from which AAARF was started - you need to be able to see these messages to ensure that AAARF is functioning properly.

Before running the animation, there are some preliminary steps you must take. First, the ring program must be started manually from the cube. This means you have to open a window on your workstation and rlogin, rsh, or telnet to the cube. AAARF connects to the cube using your userid. For the connection to work properly, you must have a .rhosts file in your login directory. If you don't already have a .rhosts file in your login directory, copy the sample rhosts file in this directory to your login directory (renaming it .rhosts). If the hostname of the workstation you are running from is not in the file, add it. (You can get the hostname of your workstation by typing "hostname" on the command line.) Edit the file to reflect your login name after each hostname entry. (This will be quite obvious when you see the contents of tne file.) Do all the editing on your file. If you don't want to edit on the cube (using vi) you can copy the file to your directory on the workstation, edit it there, and then ftp it back to the cube.

In the steps that follow, you are asked to interact with both AAARF, running on your workstation, and the ring demo, running on the cube. Once started, the cube program runs to a holding point while you interact with AAARF. Do not be concerned, no data will be lost.

To run the ring animation: (C = on the cube, W = on your workstation)

1. (C) Create the host executable (host) and the node executable (node) from he source files (host.c and node.c) to run on CX nodes (these must be copied from /usr2/aaarfDEMOS/ring to the directory of your choice on the cube):

   make cx <cr>

   AAARF does not need to know where you put the ring demo.

2. (C) Get an 8-node cube named "ring":

   getcube -c ring -t8 <cr>

3. (W) Start AAARF:

   aaarf <cr>

   If the AAARF bin directory isn't already in your path, the executables are in

   /olympus4/aaarf/bin.

   You can prepend the complete path to the name of the program if the location of the executable is not part of your current path environment variable. For example:

   /olympus4/aaarf/bin/aaarf<cr>

   will also execute AAARF. AAARF can also be run on the scgraph cluster, use:

   ~cwright/aaarf/bin/aaarf.

4. (W) Ask for the AAARF Parallel Performance Views window

   right-click/select "New Algorithm Window->Parallel Performance Views"
   in the AAARF main window

   It is recommended that new users also show the Status Display. See step (6-W) for instructions on showing the status of the animation.

Resize the window to a suitable size. You can move the animation window by dragging the frame of the window with the left mouse button. You can similarly resize the window by dragging any corner of the window with the left mouse button.

- Warning - If the diagnostic message

  "BG: server connected"

  fails to appear in the window from which you launched AAARF DO NOT PROCEED, AAARF has failed to connect to the cube. (Be patient, AAARF is launching a remote server program on the cube and waiting for it to respond - this may take up to a minuter, or more if the network, the cube, or your workstation is busy.)

- Warning - If the error message

  "BG: error binding socket: Address already in use"

  appears, the problem can be one of two things:

  (a) you have been running the ring demo and you wish to run it again - in this case the system has not released (after the last run) the socket id AAARF uses. Wait a few minutes and try again.

  (b) you haven't been running AAARF - there is nothing you can do about this one except to try another workstation. The socket id AAARF needs is being used by another process.

- Warning - If the error message

  "BG: permission denied"

  appears, there is a problem with the remote login. Check that you have copied the rhosts file into your login directory and renamed it .rhosts. Also check to ensure that you have correctly entered the hostname of your workstation, followed by your login name.

5. (C) Execute the host program.

host <cr>

the host program prompts for the number of times around the ring and the length of the message you wish to pass. The diagnostic message "aaarf_clct connecting to <your workstation>" should appear in your cube window. It may be interleaved with the input prompts, just ignore it and answer the questions. AT THE SAME TIME, the following diagnostic messages should appear in the window from which AAARF was launched:

"BG: trace socket connected"

"BG: alg socket connected"

"BG: command socket connected"

- Warning - if these diagnostic messages fail to appear, again DO NOT PROCEED, the AAARF PRASE data collection system has failed to connect with your workstation. (Once again, this may take a while - AAARF is establishing the communications link between the data collection system and the background process, PRASEBG, which is running in the background on your workstation.)

6. (W) If by some miracle you have managed to make it this far, you can start the animation by left-clicking in the animation window with the mouse or by clicking on the "GO" button in the master control panel. There may be a delay of a minute or more while AAARF catches up with your program. You can monitor the status of AAARF by left clicking on the "Status" button on the animation's master control panel. If you select the status display, note that the animation does not proceed until the "current trace time" catches up with the "next record time." It is probably best to use the Status Display the first few times you run the ring demonstration because there can be pauses in the animation while the ring programs are doing things which are not related to cube communication (such as opening files, etc.).

7. (W) When the animation finishes, kill the animation window (the status display goes with it automatically) by right-click/selecting "Quit" from the master control panel's title bar. If

you wish to run the animation again with different parameters, wait a few minutes for the system to release the socket id, and then ask for a new Parallel Performance View animation window, as before. It is not necessary to wait until the animation finishes to kill it. If you choose to kill the animation while it is still running you will have to kill the host program and the data collection program on the cube host processor. See the section "Bailing Yourself Out" below.

8. (W) Kill AAARF by right-click/selecting "Kill AAARF" in the AAARF main window's title bar.

B.4 Bailing Yourself Out

There are several situations which can lead to workstation and/or cube lockup. The most common error is trying to start the animation before the necessary communication connections have been established. Whatever the cause, here are some suggestions for extricating yourself from"terminal lockup"

First try to kill the animation window. If no menu pops up when you right click in the animation window's menu bar, try killing AAARF. If you can't kill AAARF you will probably have to go to another workstation and remotely login to proceed. In any case, the animation window and the AAARF main window may stay up on your terminal's screen. If they do, there are still AAARF related processes running which will have to be killed manually. DO NOT try to logout and log back in – this may work, or it may leave you in worse shape, and there is no way predict the outcome. Below are two command line sessions, one from the workstation, and one from the cube. Type the *ps* commands, locate the appropriate process id's, and issue the necessary *kill* command. (These are much abbreviated versions of what actually appears as a result of entering these *ps* commands.) The processes shown below may or may not show up when you issue the ps, however,

there will never be any more than what is shown below. Also, make sure that you release the cube.

```
lacertae:@~> ps -auxw | grep cwright /* workstation */
cwright 17886  R 16:34 0:00 ps -auxw
cwright 17887  S 16:34 0:00 grep cwright
cwright 17880  S 16:33 0:00 /home/hawkeye2/cwright/X/aaarf/bin/aaarf
cwright 17881  S 16:33 0:00 /home/hawkeye2/cwright/X/aaarf/bin/PPerf 5
cwright 17882  S 16:33 0:00 /tmp_mnt/home/hawkeye2/cwright/X/aaarf/bin/PRASEBG 6
cwright 17883  S 16:33 0:00 rsh cube386 /usr2/aaarf/server lacertae
lacertae:@~> kill -9 17880 17881 17882 17883

% ps -elaf | grep cwright /* cube */
10 S cwright  7194  csh -c /usr2/aaarf/server lacertae
10 S cwright  7198  /usr2/aaarf/server lacertae
10 R cwright  7200  host
10 0 cwright  7244  /bin/ps -elaf
10 R cwright  7204  /usr2/aaarf/aaarf_clct 1000000
10 S cwright  7245  /bin/grep cwright
% kill -9 7194 7198 7200 7204
```

Cube process 7200, in this case, is the ring host process. All AAARF related processes must be killed before AAARF can be run again successfully. If you restart AAARF and experience difficulties, kill it, and go back and make sure that you have removed the server and aaarf_clct programs from the cube, and the rsh program on your workstation, as well as the aaarf main program.

B.5 Instrumenting a Simple Cube Program

The AAARF data collection system, PRASE, gets trace data by intercepting certain iPSC/2 system calls, extracting the information it needs, and then passing the call on to the intended recipient. AAARF does not animate the host processor, only the node processors. However, since the host program usually controls the overall execution, it is necessary to include it in the instrumentation process. **You should be looking at the host and node listings for the remainder of this discussion.**

B.5.1 Instrumenting the host *Program*

The host program is responsible for starting the data collection program in the background.

```
#ifdef PRASE
   /* start the data collection program */
    system("/usr2/aaarf/aaarf_clct &");
#endif
```

The point at which *aaarf_clct* is started is critical: it must come after the cube has been allocated, and it should come after any interactive dialogs your program has with the user. It should also come before a *startcube()* call to prevent loss of data.

Your host program must wait until *aaarf_clct* finishes before it can finish. This is done by reading a scratch file created by *aaarf_clct* just before it finishes. Insert the declaration

```
#ifdef PRASE
    FILE *prase_ptr;
#endif
```

at the top of your main() routine, and the following code segment

```
#ifdef PRASE
    printf ("\n\nWaiting for all PRASE data to be collected.\n");
    while ((prase_ptr = fopen ("prase_end","r")) == NULL);
    system ("rm prase_end");
    fclose (prase_ptr);
#endif
```

at the end of the *main()* routine but BEFORE any calls to *killcube()*. No include files are needed.

Finally, the AAARF data collection system uses HOST_PID as the pid of the data collection program, aaarf_clct.c. If you get a compile time warning that HOST_PID is being redefined, you also are using HOST_PID for your host program and will have to change it both in the host program and any node program that communicates with the host.

B.5.2 Instrumenting the node Program(s)

The node programs are somewhat more complicated. First, add the following include file

```
#ifdef PRASE
#include "prase.h"
#endif
```

to every node program or segment which makes system communication calls (csend, crecv, isend, irecv, etc.).

For each unique node program (it should have a *main()* in it) add the following code segment at the very top of *main()*:

```
#ifdef PRASE
    prase_procs[0].num_pids = 1; /* number of processes for node 0 */
    prase_procs[0].pids[0] = 0; /* process id(s) for node 0 */
    prase_procs[1].num_pids = 1;
    prase_procs[1].pids[0] = 0;
    prase_procs[2].num_pids = 1;
    prase_procs[2].pids[0] = 0;
    prase_procs[3].num_pids = 1;
    prase_procs[3].pids[0] = 0;
    prase_procs[4].num_pids = 1;
    prase_procs[4].pids[0] = 0;
    prase_procs[5].num_pids = 1;
    prase_procs[5].pids[0] = 0;
    prase_procs[6].num_pids = 1;
    prase_procs[6].pids[0] = 0;
    prase_procs[7].num_pids = 1;
    prase_procs[7].pids[0] = 0;

    prase_lowest_node = 0;
    prase_start_time  = 0;

    praseinit();
#endif
```

If your program does not use pid=0 for the node process id's, change the pid assignments to match those which your program uses. If your system uses more than one process on any node, change the number of processes for that node and add a process id line for each additional process, making sure to assign the correct process id's.

Add the following code segment to the end of the node program:

```
#ifdef PRASE
   praseend(); /* notify aaarf_clct done */
#endif
```

There should be no system calls after this code segment, or the data will be lost. YOUR NODE PROGRAM MUST TERMINATE VIA ITS OWN ACTION. If you use *killcube()* to terminate free running node programs, they cannot be instrumented without the necessity of having to manually kill AAARF at the end of each run.

B.5.3 Changes to the Makefile

Your makefile must be changed to reflect the location of the AAARF home directory on the cube. Add the following definition:

```
AAARF  = /usr2/aaarf,
```

and update your CLAGS to define PRASE as below:

```
CFLAGS = -O -DPRASE -I$(AAARF).
```

The compilation line for the node program(s) must also include the library *aaarf_inst.a*. Libraries must be the last object module in your object module list (if you have one). For example:

```
node: node.o a.o b.o c.o $(AAARF)/aaarf_inst.a
cc $(CFLAGS) -o node node.o a.o b.o c.o $(AAARF)/aaarf_inst.a -node
```

See the makefile for the ring demo for an example. There is another example in the Carwash directory.

B.6 Source Listings

```
/*
 *  This is a modified version of the ring program, which sends a message
 *  through each node of the cube in ascending order.  The user can specify
 *  the length of the message and the number of times through the ring.  The
 *  modifications include elimination of ring count reporting and the addition
 *  of instrumentation code for AAARF.  Also, a terminate message is added to
 *  gracefully terminate the node programs so that AAARF data collection can
 *  can be completed.  Rather than kill the node processes from the host, a
 *  terminate message is passed around the ring so that each node knows when
 *  to execute praseend().  When node 0 receives the terminate message from
 *  other than the host, it sends a terminate message to the host and then
 *  quits  (All this is necessary because the ring program supplied by Intel
 *  has the node programs in a infinite loop with termination done by the
 *  host program via the killcube() command.)
 *
 *  It outputs:
 *     a) the time it took the message to go around the ring the specified
 *     number of times.
 */
char cpyright[]="Copyright (c) 1989,1990 Intel Corporation";

#include <stdio.h>

#define NODE_0      0       /* node id of node "0" */
#define NODE_PID    0       /* node process id */
#define Host_Pid    1       /* host process id */
#define ALL_NODES  -1       /* all nodes in the cube */
#define ALL_PIDS   -1       /* all process id's in the cube */
#define INIT_TYPE  10       /* type of initial message */
#define TERM_TYPE  50       /* type of terminate message */
#define COUNT_TYPE 40       /* type of count message */
#define TIME_TYPE  60       /* type of time message */
#define INIT_MSG_SIZE (sizeof(int) * 2)   /* size of initial msg in bytes */
#define TERM_MSG_SIZE (sizeof(int))       /* size of terminate msg size */
#define CNT_MSG_SIZE  (sizeof(int))       /* size of count msg in bytes */
#define TIME_MSG_SIZE (sizeof(long))      /* size of time msg in bytes */

int msg_len,            /* length of message */
    i,                  /* counter */
    ring_count,         /* # of completed ring circuits */
    term_buf,           /* buffer for terminate message from node 0 */
    msg_buff[2];        /* message buffer */

long    time_buf;       /* buffer for time information */
float   ring_time;      /* time to go around ring */
char    CR = 13;        /* ASCII Carriage Return code */

main()
{
#ifdef PRASE
```

```
    FILE *prase_ptr;
#endif
    printf("\nNumber of times through the ring (0 or neg. value quits): ");
    scanf ("%d", &ring_count);
                    /* quit the program if 0 or neg.  */
    if (ring_count <= 0)
        exit(0);
                    /* get length of msg */
    do {
        printf("Length of Ring message in bytes (0-65536): ");
        scanf ("%d", &msg_len);
    } while ((msg_len < 0) || (msg_len > 65536));
#ifdef PRASE        /* start the data collection program */
    system("/usr2/aaarf/aaarf_clct &");
#endif
    printf("\nLoading the cube\n");
    setpid(Host_Pid);
    load("node", ALL_NODES, NODE_PID);
                    /* set # of circuits to be sent */
    msg_buff[0] = ring_count;
                    /* set message length to be sent */
    msg_buff[1] = msg_len;
                    /* send message INIT_TYPE from
                     * msg_buff (length, # circuits)
                     * of length INIT_MSG_SIZE
                     * to NODE_0  at NODE_PID */
    csend(INIT_TYPE, msg_buff, INIT_MSG_SIZE, NODE_0, NODE_PID);
                    /* receive message TIME_TYPE (time of circuits)
                     * into time_buf of TIM_MSG_SIZE bytes */
    crecv(TIME_TYPE, &time_buf, TIME_MSG_SIZE);
                    /* scale time (milliseconds) */
    ring_time = (float)time_buf/1000.00;
    printf("\nRing time : %0.2f secs.\n", ring_time);
            /* send terminate message to ring */
    csend(TERM_TYPE, term_buf, TERM_MSG_SIZE, NODE_0, NODE_PID);
          /* wait for node 0 to respond */
    crecv(TERM_TYPE, &term_buf, TERM_MSG_SIZE);
          /* wait for aaarf_clct to terminate */
#ifdef PRASE
    printf ("\n\nWaiting for all PRASE data to be collected.\n");
    while ((prase_ptr = fopen ("prase_end","r")) == NULL);
    system ("rm prase_end");
    fclose (prase_ptr);
#endif
    printf("Clearing the cube\n\n");
    killcube(ALL_NODES, ALL_PIDS);
}                   /* end host program */
```

```
/*
 * This is the node program for the Ring example.
 *
 * Node 0 is the "controller" and waits for message from the host:
 *   a) the number of times to go around the RING,
 *   b) the length of the message to send around.
 *
 * It then sends a message of the desired length to
 * node 1 and counts the current circuit # around the RING.
 * After each circuit node 0 sends a current ring count
 * message to the host.
 *
 * When the circuits are completed, Node 0 sends
 * the Host the total time the message spent in the ring.
 *
 * All the other nodes patiently wait for a message and
 * then dutifully pass it on to the next node in the RING.
 */
char cpyright[]="Copyright (c) 1989,1990 Intel Corporation";

#ifdef PRASE
#include "prase.h"
#endif
#define HOST_NID  myhost()          /* host node id            */
#define Host_Pid  1                 /* host process id         */
#define INIT_TYPE  10               /* type of initial message */
#define NODE_TYPE  20               /* type of node messages   */
#define TIME_TYPE  60               /* type of time message    */
#define COUNT_TYPE 40               /* type of count message   */
#define TERM_TYPE  50               /* type of terminate message */
#define INIT_SIZE (sizeof(int) * 2) /* size of initial message */
#define TERM_SIZE (sizeof(int))     /* size of terminate msg size */
#define TIME_SIZE (sizeof(long))    /* size of time message    */
#define COUNT_SIZE (sizeof(int))    /* size of count message   */
#define MAX_MSG_SIZE 65536          /* max. example message size */

int i,                      /* loop counter */
    count,                  /* tmp storage for counter variable i */
    msg,                    /* message id for isend to host       */
    ring_count,             /* number of times to go around ring  */
    msg_len,                /* length of message */
            /* message buffer    */
    msg_buff[MAX_MSG_SIZE / sizeof(int)],
    my_node,                /* node id returned by mynode()   */
    my_pid,                 /* process id returned by mypid() */
    next_node,              /* next node in ring    */
    next_pid,               /* next process in ring */
    num_nodes;
long msg_type,              /* type of message received */
     start_time,            /* clock reading at start time */
     ring_time,             /* time spent in ring */
```

```
main() {
    my_node = mynode();          /* get node number */
    my_pid  = mypid();           /* get pid */

#ifdef PRASE
    prase_procs[0].num_pids = 1;
    prase_procs[0].pids[0] = 0;
    prase_procs[1].num_pids = 1;
    prase_procs[1].pids[0] = 0;
    prase_procs[2].num_pids = 1;
    prase_procs[2].pids[0] = 0;
    prase_procs[3].num_pids = 1;
    prase_procs[3].pids[0] = 0;
    prase_procs[4].num_pids = 1;
    prase_procs[4].pids[0] = 0;
    prase_procs[5].num_pids = 1;
    prase_procs[5].pids[0] = 0;
    prase_procs[6].num_pids = 1;
    prase_procs[6].pids[0] = 0;
    prase_procs[7].num_pids = 1;
    prase_procs[7].pids[0] = 0;

    prase_lowest_node = 0;
    prase_start_time  = 0;

    praseinit();
#endif

    num_nodes = numnodes();     /*get number of nodes in cube */
    next_node = (my_node + 1) % num_nodes;   /* calc the next node # in ring */
    next_pid  = my_pid;                      /*pid of next ring node in ring */
    if (my_node == 0) {         /*for root node only */

        for (;;) {
                /* wait for message from host */
        cprobe(-1);
                /* get message type */
        msg_type=infotype();
                /* check type for TERM or INIT */
        if(msg_type == INIT_TYPE){
                        /* recv. # of circuits and msg size of
                         * INIT_TYPE  INIT_SIZE bytes to msg_buff */
            crecv(INIT_TYPE, msg_buff, INIT_SIZE);
                        /*get circuits and msg length */
            ring_count = msg_buff[0];
            msg_len    = msg_buff[1];
                /* start timing */
            start_time = mclock();
                        /* send msg ring_count times */
            for(i = 1; i <= ring_count; i++) {
                        /* send msg_buf to the ring of
                         * NODE_TYPE length msg_len to
```

```
                    * next_node pid next_pid */
                csend(NODE_TYPE, msg_buff, msg_len, next_node, next_pid);
                        /* wait to receive the message */
                crecv(NODE_TYPE, msg_buff, msg_len);
                        /* be sure that last message has been
                         * sent so that count can be modified.
              * If not, wait */
              }/* end 'for' sending messages to ring */
                        /* calculate the time for circuits */
              ring_time = mclock( ) - start_time;
                        /* send the time msg, of TIME_TYPE
                         * size TIME_SIZE to the host pid Host_Pid */
              csend(TIME_TYPE, &ring_time, TIME_SIZE, HOST_NID, Host_Pid);
        }/* end if INIT_TYPE */
        else {          /* assume TERM_TYPE */
              /* receive TERMINATE from host */
        crecv(TERM_TYPE, msg_buff, TERM_SIZE);
              /* send TERMINATE to next node */
        csend(TERM_TYPE, msg_buff, TERM_SIZE, next_node, next_pid);
              /* wait for TERMINATE from last node */
        crecv(TERM_TYPE, msg_buff, TERM_SIZE);
              /* send TERMINATE to host */
        csend(TERM_TYPE, msg_buff, TERM_SIZE, HOST_NID, Host_Pid);
              /* quit loop */
        break;
        }/* end else TERM_TYPE */
       }/* end for loop for root node 0 */
    }/* end node 0 code */
    else {                  /* all other ring nodes execute this code */
       for (;;) {
                /* wait for message from previous node */
       cprobe(-1);
                /* get message type */
       msg_type=infotype( );
                /* check for type NODE or INIT */
       if(msg_type == NODE_TYPE){
                        /* wait to receive message of NODE_TYPE
                         * into msg_buf of MAX_MSG_SIZE bytes */
             crecv(NODE_TYPE, msg_buff, MAX_MSG_SIZE);
          msg_len = infocount( );
                        /* send message on to next node in ring */
             csend(NODE_TYPE, msg_buff, msg_len, next_node, next_pid);
       }/* end if NODE_TYPE */
       else {       /* assume TERM_TYPE */
             /* send TERMINATE to next node */
       csend(TERM_TYPE, msg_buff, TERM_SIZE, next_node, next_pid);
             /* quit loop */
       break;
       }/* end else TERM_TYPE */
       }/* end for loop non-root node code */
    }/* end non-root node code */
#ifdef PRASE
```

```
  praseend();    /* notify aaarf_clct done */
#endif
}/* end node program */
```

```
#
# Makefile for building C host and node applications for the Ring demo.
#

help:
	@echo
	@echo "You must specify the type of node you wish to build a node"
	@echo "executable for, choose one of the following:"
	@echo
	@echo "    make cx    (for 386 nodes with 387 coprocessors)"
	@echo "    make sx    (for 386 nodes with SX coprocessors)"
	@echo "    make rx    (for i860 nodes)"
	@echo

AAARF  = /usr2/aaarf

CFLAGS = -O -DPRASE -I$(AAARF)

cx:    host node    #Use default compile and link flags

sx:
	make host
	make "CFLAGS = -O -DPRASE -I$(AAARF) -sx" "LDFLAGS=-sx" node

rx:
	make host
	make "CFLAGS = -O -i860" "LDFLAGS=-i860" node

host:    host.o
	cc $(CFLAGS) -o host host.o -host

node:    node.o $(AAARF)/aaarf_inst.a
	cc $(CFLAGS) -o node node.o $(AAARF)/aaarf_inst.a $(LDFLAGS) -node

clean:
	rm host node host.o node.o
```

Bibliography

1. Bailor, Paul D. Personal Communications. Wright-Patterson AFB, Dayton OH, November 1992.
2. Brown, Marc H. *Algorithm Animation*. Cambridge, Massachusetts: The MIT Press, 1987.
3. Diane T. Rover. *Visualizing the Performance of SPMD and Data-Parallel Programs*. Technical Report, Lansing, MI: Michigan State University, August 1992.
4. Fife, Keith C. *Graphical Representation of Algorithmic Processes*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, 1989.
5. Fife, Keith C and Edward Williams. *The AAARF Programmer's Guide*.. Air Force Institute of Technology, December 1990.
6. Fife, Keith C and Edward Williams. *The AAARF Users's Guide*.. Air Force Institute of Technology, December 1990.
7. Francioni, J and Diane T. Rover. "Visual-Aural Representations of Performance for a Scalable Application Program." *Proceedings of Scalable High-Performance Computing Conference*. 433 - 440. 1992.
8. Geist, G. A., et al. *PICL: A Portable Instrumented Communication Library*. Technical Report, Mathematical Sciences Section, Oak Ridge National Laboratory, 1992.
9. Heath, Michael T. "Visual Animation of Parallel Algorithms for Matrix Computations." *Proceedings of the Fifth Distributed Memory Computing Conference*. 1990.
10. Heller, Dan. *XView Programming Manual*. Sebastopol CA: O'Reilly & Associates, Inc, 1991.
11. Hotchkiss, Robert S and Cheryl L Wampler. "The Auditorialization of Scientific Information." *Proceedings of Supercomputing '91*. 453 - 461. 1991.
12. Kernighan, Brian W. and Dennis W. Richie. *The C Programming Language*. MA: Prentice Hall, Inc, 1988.
13. Lack, Michael D. A. *Enhanced Graphical Representation of Parallel Algorithmic Processes*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1991.
14. MasPar Computer Corporation. *MasPar MP-1 Hardware Manual*, September 1990.
15. Naps, Thomas L. "Algorithm Visualization in Computer Science Laboratories," *SIGCSE Bulletin*, *22(1)*:105-110 (1990).
16. Open Software Foundation, Englewood Cliffs, New Jersey. *OSF/Motif^{TM} Programmer's Guide*, 1990.
17. Raalte, Thomas Van, editor. *XView Reference Manual*. Sebastopol CA: O'Reilly & Associates, Inc, 1991.
18. Rubin, Robert v., James Walker II and Eric Golin. "Design and Implementation of Programming Environments in the Visual Programmers Workbench." *Proceedings of the 14th Annual International Computer Software and Applications Conference*. 547-554. Piscataway, NJ: IEEE Press, 1990.
19. Shimomura, Takao and Sadahiro Isoda. "Linked-List Visualization for Debugging," *IEEE Software*, *8*(3):44-51 (May 1991).
20. Stasko, John T. "Simplifying Algorithm Animation with TANGO." *Proceedings of the 1990 IEEE Workshop on Visual Languages*. 1-6. Piscataway, NJ: IEEE Press, 1990.
21. Sun Microsystems, Inc. *Network Programming*, 1990.

22. Sun Microsystems, Inc. *OpenWindows Version 2 Release Notes*, 1990.

23. Sun Microsystems, Inc. *Programming Utilities and Libraries*, 1990.

24. Sun Microsystems, Inc. *SunOS Reference Manual*, 1990.

25. Sun Microsystems, Inc. *SunView Programmer's Guide*, 1990.

26. Sun Microsystems, Inc. *SunView System Programmer's Guide*, 1990.

27. SunSoft, Div of Sun Microsystems. *OpenWindows™ Version 3 for SunOS™ 4.1.x*, 1991.

28. Wernhart, Heidemarie and Rolan Mittermeir. "The HIBOL-2 Environment: A Basis for Visual Programmin of Business Objects," *Journal of Systems and Software*, *12*(2):157-165 (May 1990).

29. Williams, Edward M. *Graphical Representation of Parallel Algorithmic Processes*. MS thesis, School of Engineering, Air Force Institute of Technology (AU), Wright-Patterson AFB OH, December 1990.

30. Williams, Edward M. Personal Communications. Los Angeles AFB, Los Angeles CA, 1992.

31. Young, Douglas A. *Window Systems Programming and Applications with Xt*. Englewood Cliffs, New Jersey: Prentice Hall, 1989.

Vita

Captain Charles R. Wright, Jr. was born on March 7, 1953 in Savannah, Georgia. He graduated from Wayland Union High School in Wayland, Michigan in 1971. He entered the Air Force on July 21, 1971. He received a Bachelor of Science in Electrical Engineering from New Mexico State University in May of 1986. He received a commission upon graduation from Officers Training School in August of 1986, and was assigned to the Air Force Materials Laboratory, Wright-Patterson AFB, Ohio where he worked as a computer research scientist with the Manufacturing Research Group. He entered the School of Engineering, Air Force Institute of Technology in May, 1991. He graduated with a Masters in Computer Science in December, 1992.

Permanent address: 411 Round Lake Road
Caledonia, Michigan 49316

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1992	Master's Thesis

4. TITLE AND SUBTITLE
X-AAARF An X Windows Version of the AFIT Algorithm Animation Research Facility

5. FUNDING NUMBERS

6. AUTHOR(S)
Charles R. Wright, Jr., Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Air Force Institute of Technology, WPAFB OH 45433-6583

8. PERFORMING ORGANIZATION REPORT NUMBER
AFIT/GCS/ENG/92D-22

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)
WL/AAAA-2, Model Based Vision Laboratory
Target Recognition Branch, Mission Avionics Division,
Avionics Directorate, Wright Laboratory
Wright-Patterson AFB, OH

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Abstract

The AFIT Algorithm Animation Research Facility was developed by the Air Force Institute of Technology (AFIT) as a teaching aid for data structures and algorithm design of sequential processes. However, AAARF's unique design makes it particularly suitable for the animation of algorithms running on remote systems. In particular, an extensive set of parallel performance animations has been developed for the Intel iPSC/2 Hypercube for parallel program analysis and performance optimization.

The AAARF system was originally developed using the Sun Microsystems' SunView^{TM} windowing system. Recent advances in Graphical User Interface (GUI) technology combined with Sun's adoption of the X Window System as their workstation window environment, has necessitated the replacement of AAARF's GUI with a modern, X-based user interface.

This report describes the GUI replacement process, starting with selecting a GUI toolkit, designing and implementing the new user interface, testing, and finally the results of implementing the new user interface. Also included is a discussion of several changes/enhancements to AAARF which were necessary before the GUI replacement process began.

14. SUBJECT TERMS
Algorithm Animation, Visualization, X Windows System, XView, Parallel Processing

15. NUMBER OF PAGES
112

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
UNCLASSIFIED	UNCLASSIFIED	UNCLASSIFIED	UL